VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Webové služby a servisně orientovaná architektura v jazyku Java

Web Services and Service Oriented Architecture in Java Programming Language

2013 Martin Kocurek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Kocurek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Webové služby a servisně orientovaná architektura v jazyku Java
Web Services and Service Oriented Architecture in Java Programming Language

Zásady pro vypracování:

Diplomová práce je rozdělena do dvou částí. První část je zaměřena na popis realizace webových služeb v jazyce Java a popis existujících způsobů hostování takovýchto služeb. Dále je tato část práce zaměřena na vytvoření architektury šablonové služby, jež bude sloužit jako základna pro snadnou realizaci webové služby a uvedení do provozu. Druhou část diplomové práce tvoří popis problematiky servisně orientované architektury a popis způsobu použití v jazyce Java. Závěr druhé části diplomové práce bude věnován ukázce praktické realizace servisně orientované architektury v jazyce Java, jež plně využívá možností dříve vytvořené šablonové služby.

Vlastní řešení diplomové práce lze rozdělit:
1. Úvod do problematiky webových služeb v jazyce Java.
2. Servisně orientované architektury v jazyce Java.
3. Možnosti hostování webových služeb v jazyce Java a způsob realizace.
4. Analýza vlastního řešení šablonové služby.
5. Praktická realizace šablonové služby s popisem realizace.
6. Analýza ukázkového příkladu realizace servisně orientované architektury.
7. Praktická realizaci příkladu servisně orientované architektury s popisem realizace.

Seznam doporučené odborné literatury:

1.SPELL, Brett. Java : Programujeme profesionálně. Praha : Computer Press, 2002. 1022 s. ISBN 80-7226-667-5.

2.ERL, Thomas. SOA Servisně orientovaná architektura : Kompletní průvodce. Computer Press, 2009. 672 s. ISBN 978-80-251-1886-3, EAN: 9788025118863.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Hrubý**

Datum zadání: 18.11.2011

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

7. 5. 2013

Martin Kocurek

Poděkování:
Děkuji svému vedoucímu, Ing. Davidu Hrubému, za osobní přístup, odborné vedení mé práce, cenné rady, pomoc a čas, který mi věnoval.

Abstrakt

Tato diplomová práce se zaobírá tématikou webových služeb a servisně orientované architektury (SOA) v jazyku Java, které dnes tvoří důležitou součást v oblasti informačních technologií. Tato práce si klade za cíl seznámit se s principy webových služeb a SOA, s jejich technologiemi a platformou pro vývoj aplikací v jazyku Java. V další části se práce zabývá možnostmi hostování webových služeb a způsobu realizace, s důrazem na hostování v operačních systémech. Dále pak v praktické části pak ukázkový příklad implementace konkrétního a šablonového řešení webové služby. V poslední kapitole se práce věnuje SOA a praktické implementaci příkladu, který využívá předchozích možností služeb.

Klíčová slova: Java, JAX, SOA, SOAP, Webová služba, WSDL, XML

Abstract

This Diploma thesis describes theme Web Services and Service Oriented Architecture (SOA) in Java Programming Language, which today forms an important part in information technology. This thesis behind aim acquaint with principles of web services and SOA, their technology and application development platforms in Java. The next part of the thesis deals with the possibilities of hosting web services and the method of implementation, with emphasis on the hosting in operating systems. In the practical part of the thesis example of implementing specific web service solution and template web service solution. The last part of the thesis deals with SOA and the practical implementation example, which uses the previous services capabilities.

Key-words: Java, JAX SOA, SOAP, Web Service, WSDL, XML

Seznam použitých zkratek a symbolů

API	- Application Programming Interface
HTTP	- Hypertext Transfer Protocol
JAX	- Java Api for XML
JVM	- Java Virtual Machine
J2EE	- Java 2 Enterprise Edition
JAR	- Java Archive
OS	- Operating System
RPC	- Remote Procedure Call
SOA	- Service Oriented Architecture
SOAP	- Simple Object Access Protocol
UDDI	- Universal Description, Discovery and Integration
UML	- Unified Modeling Language
URL	- Uniform Resource Locator
W3C	- World Wide Web Consortium
WAR	- Web Application Archive
WS	- Web Service
WSDL	- Web Services Description Language
WSIL	- Web Service Inspection Language
XML	- Extensible Markup Language
XSD	- XML Schema Definition

Obsah

Seznam obrázků

1 Úvod

V dnešní době se většina informačních technologií stěhuje do prostředí webu, čímž se zajišťuje maximální dostupnost služeb a maximalizuje se tak počet uživatelů a tím i potenciál celé aplikace. Informační technologie, jakkoliv se od sebe jedna od druhé odlišují, jsou nyní postupně svazovány k sobě. Základním pojítkem, které toto umožňuje je právě SOA, jelikož trendy v IT systémech se jednoznačně orientují na samostatné služby a jejich skládání.

Cílem této práce je nastínit teoretické i praktické aspekty webových služeb a servisně orientované architektury, ať už pomocí platformy Java nebo obecně. Je kladen důraz popsat jednak teoretické postupy webových služeb a SOA, tak i praktické ukázky a příklady v programovacím jazyce Java. Snahou je vytvořit čtenáři dokonalý obraz toho, jak používat webové služby, na co si dávat pozor a jak pomocí nich navrhnout servisně orientovanou architekturu. Pro snadné pochopení této problematiky je text obohacen vysvětlujícími obrázky a konkrétními částmi kódů, které poskytují jasný přehled o tématu.

Uvedení webových služeb do širšího kontextu, jejich základní principy a vlastnosti, popisuje druhá kapitola. Rovněž se zabývá všemi důležitými pojmy, které musíme znát, než začneme uvažovat o nějaké smysluplnější práci. Zároveň zde najdeme teoretický popis technologií, bez kterých nemůže žádná služba fungovat. Jedná se především o XML, SOAP a WSDL. To vše zatím nezávisle na platformě a programovacím jazyce.

Na webové služby plynule navazuje třetí kapitola. Podrobně se seznámíme s možnostmi jazyka Java v kontextu s vývojem služeb. Zjistíme, že základ položila technologie JAX-RPC, která postupným vývojem přešla do v dnešní době více používané technologie JAX-WS. První konkrétní implementační ukázky srovnávají obě tyto možnosti.

Čtvrtá kapitola nabízí pohled na možnosti hostování webových služeb v jazyku Java. Soustředí se na jednotlivé případy a z čistě praktického a nezávislého hlediska nás seznamuje s nasazením služeb na webový server pomocí příkazového řádku. Najdeme zde zajímavý pohled jazyka Java na hostování služeb v operačním systému a konkrétní příklad realizace pomocí nástroje Java Service Wrapper.

Praktická pátá kapitola se věnuje implementaci šablonové služby. Od demonstrativního konkrétního příkladu přecházíme k obecnějšímu řešení, přičemž musíme myslet na efektivní implementaci komunikace a přenosu zpráv. Zde se seznámíme s technologií Castorových tříd a XML schémat a jejich použití na platformě Java.

Poslední kapitola této diplomová práce se zabývá samotnou servisně orientovanou architekturou. Seznámíme se zde se základními vlastnostmi SOA a s možnostmi implementace v Javě. Navážeme na předchozí poznatky a rozšíříme je. Konec bude patřit návrhu praktického příkladu, který za použití SOA elegantně vyřešíme a naprogramujeme.

Závěr se zabývá shrnutím, jaký význam má tato práce a jakých poznatků a implementačních výsledků jsme v jednotlivých částech dosáhli. Celkové zhodnocení obohatíme o význam celé SOA architektury a zamyšlením nad její budoucností.

2 Webové služby

V této kapitole si uvedeme obecný úvod do problematiky webových služeb. Pro další části této práce a její praktické ukázky a aplikace je nezbytně nutné rozumět základním pojmům a prvkům této oblasti a vytvořit si obrázek o tom, jak webové služby fungují a jak jednotlivé prvky služby spolu souvisí a komunikují.

2.1 Co je to webová služba

Toto slovní spojení bude hojně používáno v diplomové práci, tudíž si musíme vysvětlit, co si vlastně pod webovou službou (WS) představit. Představme si pohled běžného uživatele. Pak WS definujeme jako webové API umožňující komunikaci dvou aplikací přes síť. To vše bez ohledu v jakém programovacím jazyce je napsáno a pro jakou platformu je určeno. Komunikace mezi WS probíhá na principu zasílání zpráv, které jsou ve formátu XML. Z laického pohledu by měl tento stručný popis stačit, ale jelikož je WS celosvětově rozšířené API, má přesně definován svůj standart.

Konsorcium W3C [1] definuje formálně WS takto:

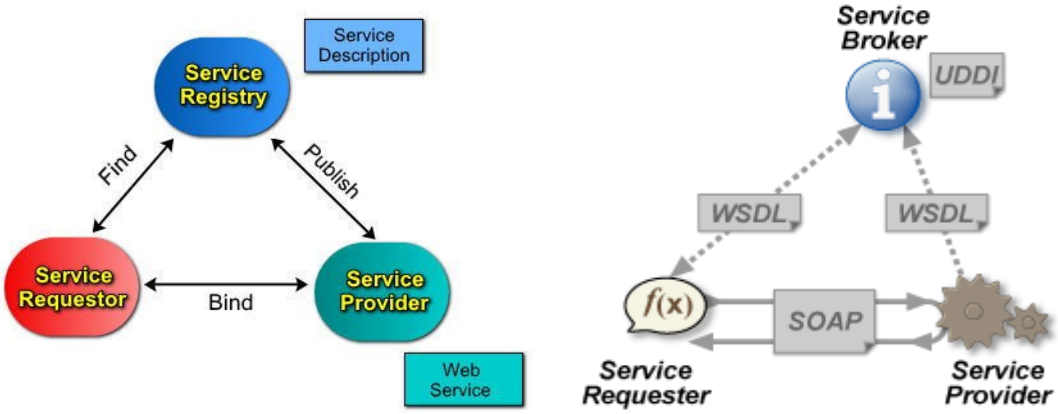
„Webová služba je softwarový systém podporující spolupráci mezi dvěma stroji prostřednictvím počítačové sítě. Rozhraní webové služby je popsáno pomocí strojově zpracovatelného formátu (přesněji formát WSDL). Ostatní systémy komunikují s webovou službou v souladu s předepsaným WSDL rozhraním s použitím SOAP zpráv typicky pomocí protokolu HTTP s XML serializací v součinnosti s dalšími webovými standardy.“

Architektura WS:

Pochopení základní architektury WS je důležitý. Na obrázku níže [2] vidíme jednoduchý model a vzájemné propojení tří rolí:

- poskytovatel webové služby (*Service Provider*) – ten poskytuje služby klientům
- klient (*Service Requester*) – vyhledává požadované služby a využívá je
- adresářové služby (*Service Registry*) – umožňují registraci, vyhledání a identifikaci služby

V typickém případě tedy poskytovatel provozuje určitou síťově dostupnou službu. Vytvoří popis této služby a publikuje jej v rámci adresářové služby nebo přímo danému klientovi. Klient na základě požadavku vyhledá (buď lokálně, nebo ve veřejném adresáři webových služeb) a identifikuje vyhovující službu, se kterou naváže komunikaci.



Obrázek 1: Architektura webových služeb, převzato z [1]

V tomto obecném úvodu do webových služeb a její architektury jsme se zatím jen okrajově zmínili o hlavních prvcích, ze kterých se webové služby skládají. Proto si v následujících kapitolách tyto technologie WS popíšeme, jelikož každá z nich tvoří důležitou funkci v principu WS, a budeme se jimi po celou dobu této práce zabývat.

2.2 Technologie WS

Základním stavebním kamenem webových služeb jsou technologie, které zajišťují přenos zpráv, popis rozhraní služeb, rejstříky a vyhledání WS. Konkrétní hlavní prvky služeb si detailněji rozebereme a ukážeme zde mezi nimi souvislosti.

2.2.1 XML

Jedná se o obecný značkovací jazyk, který byl vyvinut a standardizován pod hlavičkou W3C [3]. Obsahuje množinu pravidel a sémantických značek (atributy, elementy, jmenné prostory) a umožňuje rozdělit dokument na části dle struktury nebo identifikovat jednotlivé části dokumentů. Dále se XML může používat jako jazyk pro popis jazyka (definice syntaxe jiného jazyka). Mezi hlavní výhody patří jednoduchost, podpora v programovacích jazycích a doplnění dat o sémantiku. XML je metajazyk, tudíž jednotlivé názvy elementů nebo atributů si tvůrce dokumentu tvoří sám. Jazyk se stal základem různých sdružených technologií, které ale zde již nebudeme popisovat, jelikož v tématu WS si vystačíme s klasickým XML.

Aplikací XML (jazyk založený na XML) je celá řada, k naší práci ale hlavně potřebujeme protokol SOAP, který má v oblasti WS velký význam a bez něj se neobejdeme.

Následující část kódu ukazuje jednoduchou ukázku, jak by mohl XML soubor vypadat.

```
<?xml version="1.0" encoding="windows-1250"?>
<!DOCTYPE katalogPC "katalogPC.dtd">
<produkt>
    <název>HP ProBook 4520s</název>
    <typ>notebook</typ>
    <cena>
        <bezDPH>13 250</bezDPH>
        <sDPH>15 900</sDPH>
        <měna>Kč</měna>
    </cena>
    <popis>Tento notebook je od firmy HP.</popis>
</produkt>
```

Příklad 1: Ukázka XML souboru

2.2.2 SOAP (Simple Object Access Protocol)

Tento protokol je základním prvkem pro posílání zpráv webovým službám. SOAP umožňuje nadefinování struktury zpráv a ty si pak aplikace mezi sebou vymění. Jak je zřejmé, struktura zpráv je stanovena ve formátu jazyka XML. Samotný protokol patří do aplikační vrstvy. Při samostatném přijímání a odesílání jiných dalších zpráv využívá většinou další protokoly z této vrstvy, např. HTTP. Z historického pohledu byl protokol vyvinut firmou Microsoft a v roce 2000 vznikl SOAP ve verzi 1.1. Tato specifikace byla přijata konsorciem W3C. Dnešní aktuální verze je 1.2. [4]

Skladba SOAP zprávy:

XML jazyk nebyl zvolen náhodně jako základní kámen pro stanovení definice a struktury SOAP zprávy. Hlavním důvodem je snadná a lehká transformace XML souboru do tvaru, který potřebujeme pro konkrétní aplikaci využívající SOAP protokol v rámci komunikace. Syntaxe a pravidla XML sebou přinášejí klady ve snadné čitelnosti zprávy obyčejným uživatelem. Další plus je možnost lehce validovat samotné tělo zpráv, které nám pomůže předejít výskytu chyb při zpracování. Na druhou stranu XML syntaxe není moc vhodná z pohledu objemu dat, která přenášíme. Rychlost a kvalita komunikace je v přímém porovnání s jinými způsoby distribuovaného přenosu nedostatečná. Ty totiž posílají zprávy v binárním zápisu.

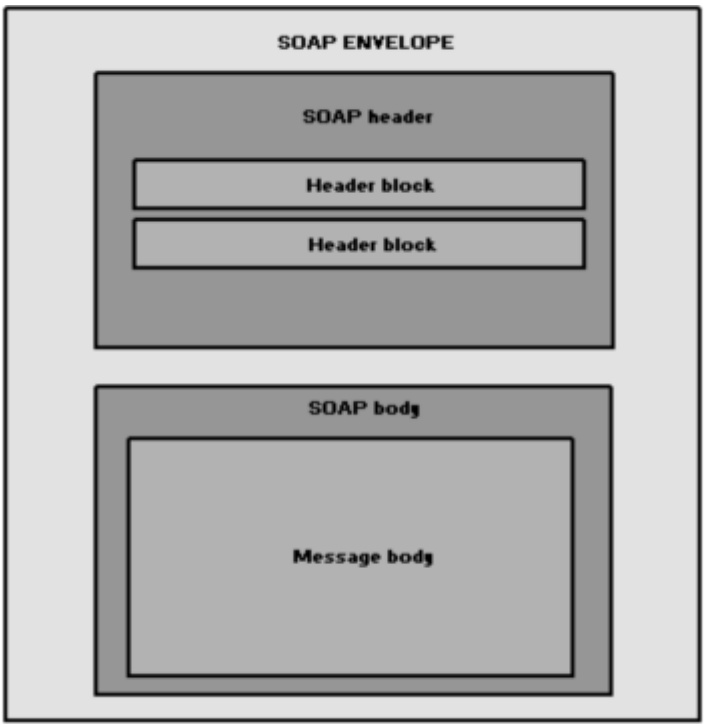
Architektura SOAP: [5]

- **Obálka (envelope)** – hlavní kořenový prvek SOAP zpráv. Definuje jmenné prostory (*namespaces*). Ty jsou schopny rozlišit elementy struktury SOAP (*headers*, *body*) od

prvků popisující samotný obsah zpráv. Dále se v ní vyskytují další dvě důležité části, které si popíšeme v následujících odstavcích.

- **Hlavička (header)** – je hned prvním prvkem obsaženým v obálce. Hlavním účelem je popis takzvaných transakčních elementů. Tyto elementy nadefinují kroky, které je nutné udělat dříve před provedení samotného těla zprávy (např. konec transakce, kontrola práv atd.). V momentě, kdy je nastavena položka transakčního elementu *mustUnderstand* na hodnotu `1`, server akceptuje tuto výzvu a provede okamžité zpracování této položky. Server zprávu může odmítnout, pokud nerozumí požadavku. Taky je možné pro starší klienty nadefinovat atribut *mustUnderstand* na hodnotu `0`. Pokud ale nastavíme hodnotu *mustUnderstand* na `1` a server nemá podporu operace, je zavolána zpráva, která vydefinuje chybu v těle zprávy. Klient je tak informován o chybě a má všechny informace potřebné k nápravě. Elementy v hlavičce můžeme doplnit o atribut *actor*, který slouží k nadefinování adresy uzlu. Hodí se to především v situacích, kdy chceme vymezit, pro které uzly jsou určeny příslušné definice hlavičky. Zpráva totiž při zpracování může procházet více uzly a docházelo by k nepřehledné situaci.

- **Tělo (body)** – obsahuje všechny důležité informace, které poskytnou provedení některé metody poskytované serverem. Dává nám fakta o jménu operace a případné parametry pro vstup do operace. Pokud se vyskytne zpráva jako odpověď, ukáže nám XML data jakožto výsledek dané operace.



Obrázek 2: Architektura SOAP, převzato z [6]

Na výše uvedeném obrázku 2 ještě jasněji vidíme strukturu SOAP architektury a jejich tří částí. Zpráva je ve formátu XML, takže si ukážeme obecnou ukázku tohoto, jak by mohl daný soubor vypadat. Vidíme SOAP zprávu, která obsahuje obálku a v ní hlavička a tělo zprávy.

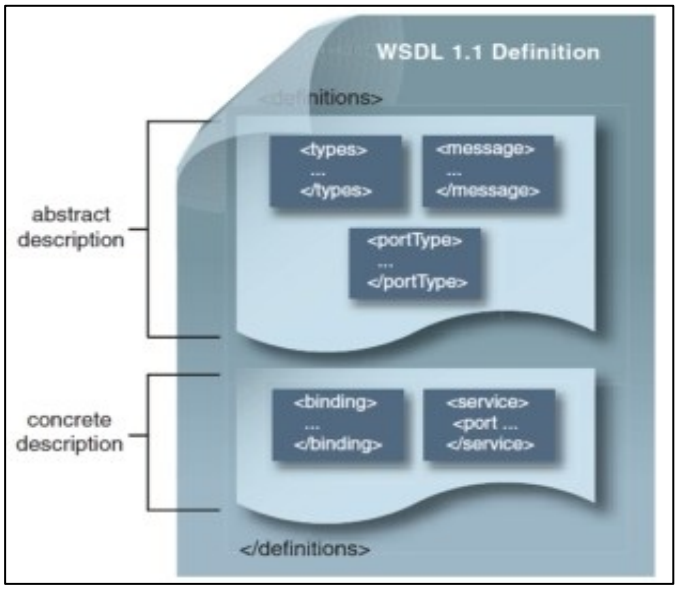
```
<soap:Envelope
      xmlns:soap="http://www.w3.org/2003/05/soap-envelope" ...>
      <soap:Header>
            <wsse:Security ...>
                  <wsse:BinarySecurityToken ...>
                  ...
                  </wsse:BinarySecurityToken>
                  <ds:Signature>
                  ...
                  </ds:Signature>
            </wsse:Security>
      </soap:Header>
      <soap:Body ...>
            <po:poNumber xmlns:po="">
                  123456
            </po:poNumber>
      </soap:Body>
</soap:Envelope>
```

Příklad 2: SOAP zpráva ve formátu XML

2.2.3 WSDL (Web Services Description Language)

WSDL je další částí, která je důležitým stavebním prvkem pro nadefinování WS. Je to jazyk sloužící pro popis rozhraní webových služeb postavený na jazyku XML jako SOAP. Toto rozhraní informuje o operacích, jaké služba poskytuje, a také o popisu komunikace, pomocí jakých portů a adres je možné ji dosáhnout. Podobně jako SOAP i WSDL spadá pod konsorcium W3C. Jazyk WSDL je stanoven ve dvou verzích 1.1 a 2.0. Krom prostého přejmenování objektů je zřejmě nejhlavnější rozdíl ve volbě mapování operací na metody protokolu HTTP (verze 1.1 mapuje pouze na metody GET a POST). Klíčová nevýhoda verze 2.0 je v chybějící podpoře vývojových nástrojů, pomocí kterých se dá s WSDL jednodušeji pracovat. V této práci se budeme zabývat verzí 1.1, protože je rozšířenější a kód lze generovat vývojovými nástroji. [7]

Popis rozhraní služby rozdělujeme do dvou částí (abstraktní a konkrétní). Tímto rozdělením si pomůžeme k opakovanému použití některých částí definice. Před samotným rozborem a zachycením odlišností obou částí si uvedeme na obrázku 3 strukturu WSDL dokumentu, kde vidíme, jaká část se z čeho skládá a vzájemné oddělení obou částí. Každý dokument začíná kořenovým elementem *definitions*, který zahrnuje obě části dokumentu.



Obrázek 3: Definice WSDL, převzato z [8]

Definice WSDL: [5]

Abstraktní část popisuje rozhraní WS z úhlu odeslaných a přijímaných zpráv a o operacích, které WS poskytuje. Operace jsou posléze seskupeny dle rozhraní. Myšlenka této části spočívá v tom, že popis všech operací a daných zpráv je jinde, než přesný formát pro samotný fyzický přenos, na kterém je WS spuštěna. Toto oddělení je pod přísným dohledem.

Obsahuje element *types*, který definuje typy používané v WSDL dokumentu při přenosu zpráv. K definici typů se používá syntaxe XML schématu. Dalším elementem je *message*, který se může v dokumentu vyskytovat vícekrát. Každá zpráva se skládá z jedné nebo více logických částí. Zpráva typicky odpovídá nějaké operaci a obsahuje informace, které jsou potřebné k provedení této operace. Každá část má přiřazené jedinečné jméno v rámci jedné zprávy. Typ části zprávy je potom definován pomocí tzv. *(message-typing atribute*), který není nic jiného než vazba na definici typu v XML schématu.

Následuje element *operation*, definující konkrétní operaci dané služby, kterou můžeme zavolat. Každý WSDL dokument musí mít alespoň jednu operaci a rozlišujeme operaci rozhraní nebo operaci propojení (dle toho kde je operace definována). Operace rozhraní má definované jedinečné jméno v rámci rozhraní a také vstupní a výstupní zprávy, pomocí kterých s operací komunikuje. Operace propojení definuje, jakým způsobem máji být vstupní a výstupní data operace rozhraní zakódována do přenosového formátu. Protože WS se spoléhají jen na komunikaci založenou na zprávách, parametry jsou reprezentovány jako zprávy. Na závěr nám zůstává element *portType*, který definuje rozhraní služby z hlediska operací. Slouží tedy jako kontejner na všechny operace. Je jednoznačně identifikovatelný dle jména a obvykle je v WSDL dokumentu jen jeden tento element.

Konkrétní část popisuje detaily systému, které jsou nutné pro samotné úspěšné nasazení služby (dostupnost služby, formát zpráv, síťová adresa, rozhraní z abstraktní části).

Obsahuje element *binding*, jehož hlavním účelem je propojit element *portType* z abstraktní části. Jinými slovy, tento element představuje jednu z možných technologií přenosu, kterou služba může použít pro komunikaci. Protokol SOAP je nejběžnějším typem vazby, ale jsou podporovány i jiné. Vazba se aplikuje na celé rozhraní nebo jen na určitou operaci. V jednom WSDL dokumentu může být více těchto elementů.

Závěrečným elementem konkrétní části je element *service*. Jeho role je v zabezpečení síťové adresy konkrétnímu *binding* elementu. Je jednoznačně identifikovatelný dle svého jména a krom toho obsahuje také elementy *port*, které definují fyzickou adresu (koncový bod). Cože je množina výstupních bodů, na kterých je daná služba poskytována.

Uvedli jsme si seznam základních komponent WSDL dokumentu. Kompletní dokumentaci a další komponenty můžete najít zde [8]. Každý WSDL dokument, který je validní vůči svému XML schématu a splňuje podmínky kladené na definici jednotlivých elementů (komponent), lze prohlásit za dokument WSDL odpovídající specifikaci WSDL 1.1/2.0.

2.2.4 XML Schéma

V souvislosti s WSDL si uvedeme ještě popis XML schématu, jakožto popis struktury XML dokumentu. Opět jako všechny tyto standardní technologie spadá schéma pod konsorcium W3C. Tato technologie se dotýká WS značným způsobem, protože je dokonce konsorciem doporučeno pro tvorbu výše uvedených WSDL dokumentů. Schéma se podílí na tom, jaké data vystupují ve zprávách a jak jsou strukturovány.

Schéma definuje: [9]

- strukturu XML dokumentu
- elementy a atributy v XML dokumentu
- dětské elementy, jejich počet a pořadí
- obsah elementu (prázdný / neprázdný)
- datové typy elementů a atributů
- defaultní a pevné hodnoty elementů a atributů

Důležitou součástí WSDL dokumentu a XML schématu je celkový jmenný prostor *targetNamespace*, který zapouzdřuje jednotlivé elementy, atributy a typy do daného jmenného prostoru a tím pádem je možné se vyhnout potenciálním kolizím. Více se o schématech dozvíme v praktické části při přenosech zpráv v kapitole 5. Propojíme je s programovacím jazykem a dokážeme z dokumentu XSD vytvořit třídy v jazyce Java.

2.2.5 UDDI (Universal Description, Discovery and Integration)

UDDI nabízí mechanismy pro registrování, kategorizování a vyhledávání webových služeb. UDDI funguje jako velký adresář, který obsahuje informace o subjektech (firmách) a jimi poskytovaných službách. Samotný registr pracuje rovněž jako webová služba a komunikace s ní tedy opět probíhá pomocí protokolu SOAP. Služba není pod standardem W3C, ale pod hlavičkou OASIS.

UDDI registr obsahuje následující čtyři druhy entit: [10]

- **podnikatelské entity (firmy)** – business entity, u každé firmy v registru jsou zaznamenány základní údaje jako název, stručný popis a kontaktní údaje. Každé firmě mohou být přiřazeny klasifikační identifikátory, které určují oblasti jejího podnikání a geografickou polohu.

- **služby** – business service, ke každé firmě jsou v registru uloženy seznamy služeb, které firma poskytuje. Každá služba je opět popsána a obsahuje seznam šablon vazeb, které ukazují na technické údaje nutné pro využití služby.

- **šablony vazeb** – binding template, šablony popisují, jak a kde je možné se službou komunikovat. Typicky je tato informace popsána odkazem na WSDL soubor s definicí rozhraní služby. Každá šablona kromě toho odkazuje na typ služby, který implementuje.

- **typy služeb** – service typ, typ služby definuje abstraktní službu. Funguje tedy jako obdoba rozhraní, jak je známe např. z Javy. Několik firem může nabízet stejný druh služby se stejným rozhraním a tedy i typem služby. Typ služby je popsán tzv. technickým modelem.

Typická práce s UDDI probíhá tak, že vývojář prohledá registr a najde si služby, které potřebuje. Získá pro ně popis WSDL a může je začít rovnou používat. Dodejme ještě, že UDDI nemusí obsahovat jen popisy webových služeb ve WSDL. Lze do něj ukládat popisy služeb v libovolném formátu.

2.3 Výhody a nevýhody WS

Jako každá technologie, tak i webové služby mají své klady, ale také své zápory. Proto si zkusme tyto vlastnosti shrnout do několika bodů. [5]

Výhody:

- nezávislost na programovacím jazyce a na platformě
- vychází z výhod využití XML

- bezproblémová lokalizace
- značná nezávislost klienta a poskytovatele služby
- snadná integrace v rozsáhlých projektech
- SOAP a WSDL jakožto silný nástroj pro podnikové aplikace

Nevýhody:

- poměrně značná rozsáhlost dat
- zpracování dat (XML) je pomalé a drahé
- složitost a nepřehlednost specifikací WS
- komplikované využití podle SOAP a WSDL z pohledu webových aplikací

2.4 Využití WS a význam vůči SOA

Po přečtení této kapitoly si přesně dokážeme představit, co webové služby vlastně jsou. Ještě si ale velmi stručně shrňme, k čemu se hlavně WS užívají, a v jakých oblastech můžeme najít i příslušné výše popisované technologie.

Webové služby byly vyvinuty, a jejich hlavní účel je právě v této oblasti, jako implementace SOA (*Service Oriented Architecture*). Základní myšlenka SOA je rozdělení funkcí do oddělených malých nezávislých částí, které označujeme pod pojmem služba (*service*). Služby jsou pak volně provázány buď s operačním systémem, kde běží či jsou nasazeny, anebo s programovacími jazyky, na kterých jsou naimplementovány. Dostupnost je obvykle zajištěna přes síť a komunikace probíhá dle dobře nadefinovaných, osvědčených a otevřených formátů (XML). Služby jsou sice nastaveny jako nezávislé, ale poměrně často je běžné nadefinovat komplikovanější operace, které jsou schopny zavolat několik služeb po sobě v určité posloupnosti a za určitých podmínek. Takovému spojení se v SOA říká orchestrace služeb. [5]

Pokud porovnáme tyto myšlenky SOA a klasický standard WS, všimneme si, že mají hodně co společného či velmi podobného. Oddělené a nezávislé části tvoříme pomocí webových služeb také, i ony jsou přístupné pomocí síťové komunikace, a jejich nezávislost na platformě a programovacím jazyce je zcela jistě rovněž zřejmá. Webové služby tvoří hlavní oblast svého zájmu v systémech, kde je možno jednoduše nabízet a sdílet funkce ostatním systémům, aniž by nám záleželo na použitém programovacím jazyce pro jejich vývoj. Více se dozvíme o SOA v kapitole 6 této diplomové práce.

3 Webové služby v jazyce Java

Vysvětlili jsme si základní prvky a kostru webových služeb a nic nebrání k tomu se přesunout k jazyku Java a popsat si možnosti samotného vývoje. V této praktické kapitole si popíšeme metody a technologie, které nám tento programovací jazyk nabízí s malými ukázkami zdrojových kódů.

3.1 Vývoj služeb pomocí technologie JAX – RPC

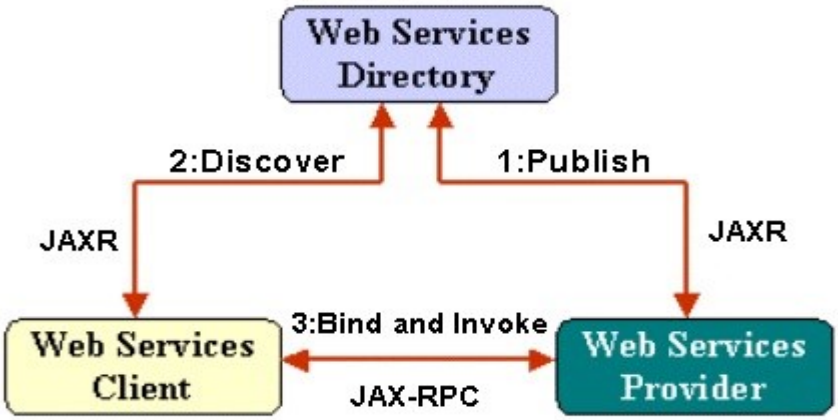
Programovací jazyk Java nám nabízí platformu J2EE. Ta byla vyvinuta pro integraci webových služeb a je jednou z mnoha možností, (pokud tedy nepočítáme jen omezení na samotnou Javu) jak služby implementovat, využít a poskytnout. Tyto komponenty mohou být snadno nastaveny jako webové služby. Poskytují mimo jiné podporu transakcí, připojení k databázi, řízení životního cyklu a mezi plusy a přednosti patří přenositelnost, škálovatelnost a spolehlivost. Nejprve se zaměříme se na starší verzi 1.4, která nabízí pro vývoj webových služeb rozhraní JAX-RPC. To nám mimo jiné poskytuje implementaci služeb založené na protokolu SOAP a také samozřejmě nabízí popis rozhraní webových služeb založených na WSDL. Jak později uvidíme, služby budou jednoduše naprogramovány a nasazeny do provozu. Pomocí této platformy jsou webové služby snadno přenositelné a mohou také spolupracovat s libovolnou webovou službu vyvinutou jiným způsobem či technologií.

Tato menší kapitola by měla objasnit krok za krokem, jak vyvíjet webové služby pomocí této platformy a ukážeme si i malé příklady, které mohou sloužit jako základ pro vývoj větších aplikací.

3.1.1 Úvod do problematiky J2EE a JAX-RPC

Pro práci s touto platformou si stručně nastíníme architekturu a princip jejího využití. Následující obrázek 4 ukazuje, jak Java API pro XML registry (JAXR) a JAX-RPC hrají důležitou roli při používání webových služeb. Z architektonického pohledu můžeme webové služby považovat za servisně orientovanou architekturu, která se skládá z více služeb, které spolu komunikují přes dobře definované rozhraní. Jednou z výhod servisně orientované architektury je, že umožňuje vytvoření volně vázaných aplikací, které mohou být distribuovány a lze k nim přistupovat z libovolného klienta v rámci celé sítě. [11]

J2EE poskytuje nástroje, které nám umožní rychle vytvářet, testovat a nasazovat webové služby. Každá WS je pak nabídnuta klientům, kteří k ní mohou přistupovat a využívat. Vytvořené aplikace pak mohou komunikovat s jinými webovými službami, bez ohledu na to, jak jsou implementovány.

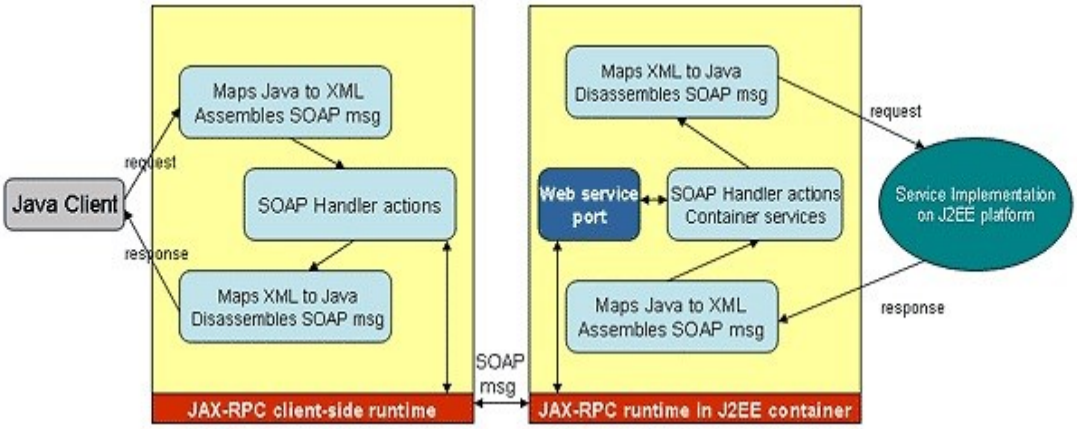


Obrázek 4: J2EE model, převzato z [11]

JAX-RPC je Java rozhraní pro XML, které je založeno na vzdáleném volání procedur jako lokálních (RPC). Můžete ho použít k vytvoření webové služby a klienta, kteří používají RPC a XML. RPC je reprezentováno použitím protokolů založený na XML. V našem případě je to SOAP, který definuje strukturu, kódování, pravidla a konvence pro reprezentaci RPC volání a odpovědí, které jsou vysílány jako SOAP zpráv přes HTTP. Výhodou JAX-RPC je i to, že skrývá složitosti SOAP zprávy pro programátora. Nyní si popíšeme a ukážeme, jak to celé spolu funguje a komunikuje.

Vývojář nastaví vzdálené procedury (v našem případě webové služby), které lze zavolat prostřednictvím vzdáleného klienta pomocí rozhraní, které naimplementujeme v programovacím jazyce Java. Klient vidí webové služby jako soubor metod, které provedou příslušné operace a logiku v jeho prospěch. Klient přistupuje k webové službě pomocí koncového bodu rozhraní služby, jak je to nadefinováno v JAX-RPC. Klienti vytvoří vývojářům rozhraní klient - proxy server (neboli místní objekt, který představuje vzdálenou službu), který je automaticky generován - a pak jednoduše zavolá metody na proxy serveru. Vývojář se nemusí starat o vytváření či parsování SOAP zpráv, to vše je zařízeno pomocí JAX-RPC systému. Je třeba si uvědomit, že J2EE webové služby mohou být zavolány libovolným klientem webové služby, ale žádní J2EE klienti webových služeb nemůžou být zavolání webovou službou. [11]

Pro lepší pochopení vidíme na následujícím obrázku, jak klient komunikuje s webovou službou vytvořenou v jazyce Java na platformě J2EE. Můžeme si všimnout, že J2EE aplikace mohou používat webové služby, které jsou zveřejněny jinými poskytovateli, bez ohledu na to, jak jsou implementovány. Jak již bylo zmíněno, všechny detaily mezi žádostí a odpovědí jsou skryty v zákulisí. Vše se zabývá pouze přímo programovacím kódem v jazyce Java, což jsou mimo jiné metody a datové typy. Nemusíte se starat o mapování Java do XML nebo budovat SOAP zprávy. To je výhodné pro vývojáře, kteří se mohou soustředit na kvalitní řešení daného problému.



Obrázek 5: Java Klient a J2EE Webové služby, převzato z [11]

Z obrázku 5 můžeme také vidět, že webový klient nikdy nepřistupuje k službám přímo, ale prostřednictvím kontejneru. To je určitě výhoda, protože webovým službám přidává navíc funkce, které sám kontejner obsahuje (kvalita služeb, lepší zabezpečení, atd.).

Před samotnou práci s JAX-RPC je třeba brát na vědomí, že se mapují datové typy v Javě na XML a WSDL dle jejich definic. Dobrou zprávou je, že nemusíme znát podrobnosti o těchto mapováních, ale měli bychom si uvědomit, že ne všechny třídy J2EE mohou být použity jako metody, parametry nebo návratové typy v JAX-RPC. JAX-RPC podporuje následující primitivní datové typy: boolean, byte, double, float, int a pole.

3.1.2 Vytvoření webové služby v Javě pomocí JAX-RPC

Celý proces a konstrukce webové služby pomocí JAX-RPC zahrnuje pět základních kroků. Důkladně se podíváme na každý tento bod podrobněji, abychom mohli nějakou vzorovou jednoduchou webovou službu vytvořit. U každého kroku zde uvedeme příklad hlavních prvků zdrojového kódu pro inspiraci k další práci.

1. Návrh kódu koncového rozhraní webové služby
2. Implementace rozhraní
3. Napsání konfiguračních souborů
4. Vygenerování potřebných souborů
5. Nasazení samotné služby

Návrh kódu koncového rozhraní webové služby:

Jedná se o první krok při vytvoření nějaké webové služby. Vytvoříme koncové rozhraní, kde se specifikuje, které metody webové služby může vzdálený klient využít. Následuje ukázka kódu a také je důležité dodržet při tomto vývoji pár věcí:

- Musí rozšiřovat rozhraní *java.rmi.Remote*
- Nesmí obsahovat definice konstanty jako *public static final*

- metody ošetřit vyjímkou *java.rmi.RemoteException*
- parametry v metodách a návratové typy musí být podporovány typy v JAX-RPC

```
public interface MathFace extends Remote {
  public int add(int a, int b) throws RemoteException;
}
```

Implementace rozhraní služby:

V dalším kroku vytvoříme třídu v Javě, která bude implementovat rozhraní a všechny metody ve výše uvedeném bodě. Z následující ukázky zdrojového kódu by mělo být vše jasné.

```
public class MathImpl implements MathFace {
  public int add(int a, int b) throws RemoteException {
    return a + b;}}
```

Napsání konfiguračního souboru:

Důležitým krokem je nadefinování konfiguračního souboru. Zde se využívá jazyk XML a samotná konfigurace se obvykle píše do souboru *config.xml*. V tomto souboru se nejčastěji popisuje název samotné služby, jmenný prostor, název balíčku a název rozhraní. V následující částečné ukázce je vše nadefinováno. Na základě těchto informací by se měl po kompilaci vytvořit WSDL soubor.

```
<service
      name="MyFirstService"
      targetNamespace="urn:Foo"
      typeNamespace="urn:Foo"
      packageName="math">
      <interface name="math.MathFace"/>
</service>
```

Toto mapování udává, že služba se jmenuje *MyFirstService*, WSDL používá výchozí jmenný prostor *urn:Foo*, atd. Nejdůležitější je však část definice *interface*, která určuje rozhraní koncového bodu a jeho implementaci. Takto vytvořenou službu lze zkompilovat a vygeneruje se dokument WSDL, popisující její rozhraní.

Vygenerování potřebných souborů:

Nyní je čas zkompilovat aplikaci, aby vývojové prostředí bylo schopno zajistit vygenerování dalších potřebných souborů. Mezi ně určitě patří popis rozhraní webové služby, WSDL soubor. Tento soubor obsahuje popis dané služby, kterou mohou klienti zavolat. Taky se musí vygenerovat *xml* soubor, který slouží k namapování XML a Javy. Struktura tohoto souboru je úzce podobná příslušnému WSDL souboru. Každá nabízená služba je zde reprezentována

jako služba rozhraní namapovaných prvků. Po tomto kroku máme již postavený základ webových služeb a je možné tyto služby nasadit do provozu. Ukázky kódu zde nebudeme uvádět, jelikož se jedná o automaticky vygenerované soubory a jejich skladbu nemusíme v tomto případě zas až tak moc znát. Navíc kostra a složení webových služeb jsou popsány v předchozí kapitole.

Nasazení webové služby do provozu:

Webové služby vytvořené pomocí JAX-RPC jsou webové komponenty a můžeme je tedy použít pro zavedení v prostředí webu. Pokud službu implementujeme ve vývojovém prostředí, můžeme zde použít testovací server pro spuštění takovéto služby. Postup jak vytvořit jednoduchou službu pomocí JAX-RPC je za námi. Nyní se budeme zabývat klienty, kteří by mohli službu využít ve svůj prospěch.

3.1.3 Vytvoření klienta v Javě pomocí JAX-RPC

Důležitou součástí každé webové služby je klient. Ten získá prostředky, aby se k službě dostal a následně pomocí JAX-RPC technologie spustí webovou službu stejným způsobem, jakoby metody volal lokálně. V Javě existují 3 typy klientů:

- **Static Stub** – třída, která je staticky vázaná na koncový bod rozhraní služby. Definuje všechny metody, které definuje rozhraní služby. Proto může klient využít metody webové služby přímo přes testovací kód. Výhodou je jednoduchý a snadný kód, naopak nevýhoda je, že každá změna ve webové službě, musí být změněna i v testovacím kódu. Tento klient se používá hlavně u služeb, které jsou stabilní a nedochází zde často ke změnám.

- **Dynamic Proxy** – podporuje koncový bod rozhraní webové služby dynamicky za běhu. Proxy server získává instance služby za provozu. Volá se stejným způsobem jako Static Stub, ale použití je výhodné u služeb, které se často mění.

- **Dynamic Invocation Interface** – v Javě je definováno v *javax.xml.rpc.Call*. Jedná se o instanci objektu dynamického volání. Na rozdíl od výše uvedených druhů klienta musí být nakonfigurován dříve, než může být použit. Klient musí poskytnout název operace, název parametrů, datové typy a port. Výhodou je, že od zavolání není na nic už vázán, tudíž ho neovlivní případné změny na straně klienta.[11]

Static Stub

Jedná se o nejčastější typ klienta, který se v JAX-RCP používá. Před samotným vytvořením tohoto klienta v Javě je nejdříve nutné zapsat konfigurační soubor (v XML), který popisuje umístění WSDL souboru, který se nám vygeneroval při vytváření služby. Jedná se hlavně o řádek s URL, kde se nachází WSDL soubor. Následuje tato ukázka a za ní hned zdrojový kód popisuje pouze kostru a důležité metody v Javě.

```
<configuration>
 <wsdl location="http://localhost:8080/MyService?wsdl"/>
</configuration>

public static void main(String[] args)
{
  try
  {
      Stub stub = createProxy();
      stub._setProperty(javax.xml.rpc.Stub.ENDPOINT,
      "http://localhost:8080/MyService/");

  }
  catch(Exception ex)
  {
     ex.printStackTrace();
  }
}
private static Stub createProxy()
{
  return(Stub)(new MyService().getPort())
}
```

Klient vytvoří proxy server, ve kterém si přiřadí příslušný port své služby a metoda *setProperty* má dva parametry, koncový bod rozhraní služby a URL adresu, kde je služba v provozu nasazena.

Dynamic Proxy

Další klient JAX-RPC, který se v praxi používá. Prvním krokem je vytvoření konfiguračního souboru. Pokud bychom ho chtěli použít pro další práci, je třeba nahradit „*dynamicproxy*“ pojmenováním našeho aktuálního balíčku. Znovu se jedná pouze o část zdrojového kódu klienta, snažíme se ukázat nejdůležitější prvky, které tento typ charakterizují.

Popis uvedeného zdrojové kódu není tak složitý jak vypadá. Jednoduše je vytvořena instance služby *Factory*. Tato služba se používá k vytvoření objektu služby, která se chová jako továrna na proxy server. Jak můžeme vidět, metoda *createService* má dva parametry: URL ze souboru WSDL a QName objekt, který představuje XML kvalifikovaný název - jmenný prostor URI a název služby. Proxy objekt je tedy vytvořen a je volána finální metoda na tento objekt.

```
public static void main(String[] args)
{
   try
   {
```

```
      String nameSpaceUri = "urn:Foo";
      String serviceName = "MyFirstService";
      String portName = "MathFacePort";¨
      URL url = new URL("http://localhost:8080/MyService?WSDL");
      ServiceFactory serviceFactory=ServiceFactory.newInstance();


      Service mathService = serviceFactory.createService(url,
                    new QName(nameSpaceUri, serviceName));

      dynamicproxy.MathFace myProxy = (dynamicproxy.MathFace)
      mathService.getPort(new QName(nameSpaceUri,
      portName),dynamicproxy.MathFace.class);

     }
     catch (Exception ex)
     {
       ex.printStackTrace();
     }
    }
}
```

3.1.4 Zhodnocení JAX-RPC

Popsané API bylo vymyšleno v rámci Java EE 1.4, a jak je zřejmé, kompatibilita je zajištěna jen na prostředí Java 1.4. Vývoj nezastavíme a tohle je hlavní problém a nedostatek, jelikož jazyk Java se neustále posouvá kupředu. Omezení v užití běžných věcí jako anotace či generické typy je limitující. V práci jej uvádíme hlavně z počátečních historických důvodů a jako příklad, kde začal a jakým směrem postupoval vývoj webových služeb v Javě. Na druhou stranu, pokud se nám hodí zprovoznit funkce pomocí WS v rámci Javy 1.4, nemáme jinou volbu než JAX-RPC.

3.2 Vývoj služeb pomocí technologie JAX – WS

Teď se podíváme na dalšího zástupce technologie služeb a tím je JAX-WS. Vyvinut byl jako nástupce předchozího popisovaného způsobu JAX-RPC. Můžeme říct, že JAX-RPC 2.0 je přepsáno na JAX-WS 2.0. Základní myšlenka je stejná a nemělo by smysl ji ani měnit, tj. chceme poskytnut rozhraní, které velmi zjednoduší vývoj WS a nebude zatěžovat Java vývojáře od zpráv typu WSDL a SOAP. Pokračovatel sebou přinesl i řadu novinek a zlepšení. Přechodem na Javu 5 vznikly nové možnosti a s nimi např. anotace a jiné výhody, které z moderní Javy známe. Zjednodušila se práce s koncovými body i samotná struktura při implementaci. Ukázky a postupy jsou uvedeny v dalších podkapitolách. JAX-WS není kompatibilní s předešlými verzemi Javy právě kvůli anotacím. Dalším faktorem je odlišný způsob a styl při zavedení koncových bodů.

3.2.1 Úvod do problematiky Java EE a JAX-WS

Následující body shrnují tuto technologii a můžeme si uvědomit i případné rozdíly mezi WS a RPC.

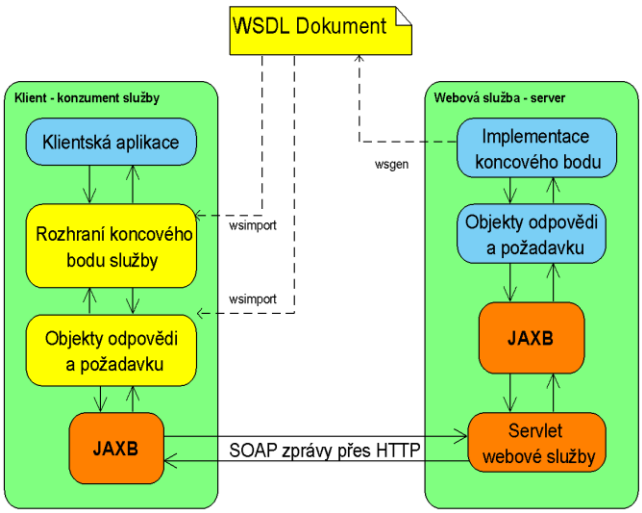
- pro realizaci XML webových služeb a jejich klientů v Java EE
- používá přenos XML zpráv SOAP protokolem přes http
- podporuje „*message-oriented*“ a „*RPC-oriented*“ webové služby
- webové služby jsou definovány jako Java třídy s poskytovanými metodami označenými pomocí anotací (automatický převod definice rozhraní třídy do popisu pomocí WSDL)
- spotřebitel vytvoří pomocí JAX-WS lokální proxy pro vzdálenou webovou službu a tu transparentně používá (proxy má stejné rozhraní jako třída implementující webovou službu u poskytovatele, převod volání metod proxy na SOAP zprávy je automatický)
- respektuje nezávislost na platformě – lze komunikovat s libovolně implementovanými službami (tzn. se službami neimplementovanými v Javě, pomocí JAX-WS, apod.) [12]

Hlavní novinky a přínosy oproti JAX-RPC:

- Již jsme zmiňovali zavedení anotací a obecně zjednodušuje implementaci kódu pro vývoj. Již nemusíme mapovat koncový bod. Zmenšilo se množství vygenerovaných tříd – menší a soudržnější aplikace (klienti i poskytovatelé služeb).
- Pozměněn přenos dat do SOAP zpráv – převedeno na JAXB (*Java API for XML Binding*), které hodně zlepšuje způsoby namapování Java objektů na XML elementy. JAXB podporuje XML schémata, oproti JAX-RPC, kde je pouze stanovena určitá část a podmnožina typů z XML schématu pomocí vlastních pravidel.
- Podpora novějších protokolů (SOAP 1.2).
- Podpora obou možností a přístupů k namapování, buď RPC styl mapování, nebo orientovaný na zprávy/dokumenty (*message/document oriented*). [12]

Princip práce JAX-WS:

V této části bychom si podrobněji popsali, jakou cestou probíhá komunikace mezi poskytovatelem služeb v JAX-WS a klientem, který službu chce využít. Komunikace je pěkně znázorněná na obrázku 6.



Obrázek 6: Komponenty JAX-WS, převzato z [13]

Obrázek ukazuje, jak je uskutečněná komunikace klienta a webové služby. Význam barev je následující:

- žluté složky – vygenerovaný kód;
- modré složky – implementují vývojáři;
- oranžové složky – to nám poskytne přímo JAX-WS API

Jako vývojáři jsme povinni naimplementovat koncový bod a dle naší potřeby případně vyspecifikovat objekty odpovědi a požadavku (viz obrázek 6). Už jen tady z tohoto jsme schopni vytvořit příslušný WSDL dokument, který reprezentuje rozhraní služby. Jako vývojáři klientské aplikace můžeme potom dodat dané rozhraní koncového bodu a příslušné objekty pro komunikaci se službou. Celý proces komunikace vypadá asi takto: [13]

- Vygenerované rozhraní koncového bodu slouží klientovi k zavolání všech metod, které WS poskytuje. Objekty požadavku jsou uvedené jako parametry.

- JAXB převede objekty požadavku do očekávaného formátu XML, který reprezentuje tělo SOAP zprávy. Toto tělo je poté zabaleno SOAP obálkou a posláno pomocí protokolu HTTP na příslušnou URL adresu WS.

- Java Servlet WS nám sám rozhodne, o kterou ze služeb se jedná (vybere vhodnou třídu, která má požadavek obsloužit). Vybalí obálku a obstará nám její tělo JAXB.

- Celý proces je ukončen převedením obsahu těla zpět z XML do formátu Java objektů. Ty jsou dodány jako vstupní parametry implementaci koncového bodu, která provede požadované operace a odpověď pošle velmi podobným způsobem jako klient svůj požadavek. [13]

Uvedli jsme si teoretické základy této technologie a ukážeme si postup vytvoření služby na konkrétní ukázce, která se zabývá hlavně komunikací klienta a webové služby, jenž jsou implementovány pomocí JAX-WS. Je však možné kteroukoliv ze stran přehodit, tj. klienta nebo WS nahradit jiným klientem a službou v jiném programovacím jazyce. To je možné právě kvůli standardu a pravidlům, který JAX-WS splňuje.

3.2.2 Vytvoření webové služby v Javě pomocí JAX-WS

Po teoretickém rozboru máme přehled, jak komunikace a celý proces běží mezi klientem a webovou službou. Proto si opět ukážeme, jakým způsobem vytvořit jednoduchou reprezentaci služby v JAX-WS a tím bude jasný posun kupředu oproti JAX-RPC. [14]

Základem je použití anotací *@javax.jws.WebService* a *@javax.jws.WebMethod*, které dříve nebyly možné, neboť až od verze Javy 5 se zavedly právě anotace a další novinky:

- webová služba je realizována jako rozhraní/třída s anotací *@WebService* (service endpoint interface/service endpoint implementation (SEI))
- implicitní je definice rozhraní služby společně s implementací (ve třídě), (přímo u implementace metod třídy pomocí anotace *@WebMethod*)
- parametr *endpointInterface* umožňuje oddělit definici rozhraní služby, (rozhraní musí rozšiřovat rozhraní *Remote* a označit metody pomocí *@WebMethod*)
- třída implementující službu nesmí být *final* nebo *abstract*, musí mít implicitní bezparametrický konstruktor, (metody s anotací *@PostConstruct* a *@PreDestroy* obslouží vznik a zánik)
- metody rozhraní/třídy pro webovou službu (anotace @WebMethod) musí být *public* a nesmí být *static* nebo *final*,
- typy parametrů a návratových hodnot metod rozhraní/třídy pro webovou službu jsou omezeny Java Architecture for XML Data Binding (JAXB),
- z třídy (tj. z implementace služby) lze generovat popis služby.

Jako i u předchozí technologie musíme prvně naimplementovat koncový bod. Ta je velmi zlehčena a stačí, jak vidíme na ukázce, pouze jedna třída anotovaná příkazem *@WebService*. Potom již definujeme pouze metody služby, kdy každá služba musí mít aspoň jednu metodu. Ty musíme ohraničit anotacemi *@WebMethod*, které popisují název operace WS. Parametry v metodách označujeme pomocí anotace *@WebParam*, které pojmenují příslušné parametry služby. Krátká ukázka zdrojového kódu koncového bodu je taková:

```
@WebService()
public class TestWS
{
   private String hello = "hello ";
   @WebMethod(operationName = "sayHello")
   public String sayHello(@WebParam(name = "name")String name)
   {
           return hello + name;}
}
```

Tento jednoduchý postup ukazuje opravdu vše, nejsou již třeba další soubory na mapování nebo rozhraní. K této službě vytvoříme generátorem nebo ručně WSDL dokument a službu vystavíme společně s WSDL na aplikačním serveru. Na tomto příkladu vidíme značný pokrok směrem vpřed oproti předešlé technologii. Shrňme si tedy vývoj služby v těchto krocích:[14]

1. tvorba kódu třídy s implementací služby
2. kompilace třídy s implementací služby
3. použití nástroje pro generování popisu služby
4. zabalení zkompilované implementace a popisu do WAR archivu
5. umístění WAR archivu na aplikační server (aplikační server automaticky generuje popis služby vyžadovaný spotřebitelem)

Celý proces můžeme usnadnit jakýmkoliv vývojovým prostředím (IDE).

3.2.3 Vytvoření klienta v Javě pomocí JAX-WS

Základem je použití anotace *@WebServiceRef*: Navíc si můžeme i tady uvědomit, k jakému pokroku a posunu ve vývoji došlo. [14]

- klient deklaruje odkaz na webovou službu anotací *@WebServiceRef*, (parametr *wsdlLocation* udává URI na WSDL odkazované služby)
- anotace se použije s deklarací proměnné zastupující objekt poskytovatele proxy, (proměnná se deklaruje jako *static* a bez přiřazení hodnoty, typ proměnné, tj. třída objektu, je třída implementující službu doplněná v názvu slovem „*Service*“
- objekt poskytovatele proxy se použije k získání proxy (portu služby), (metoda *getNamePort*, kde slovo „*Name*“ je název třídy implementující službu)
- proxy se používá jako klasický objekt třídy implementující službu. (práce s webovou službou zastupovanou proxy je plně transparentní, jako práce s lokálním objektem třídy implementující webovou službu)
- z WSDL lze generovat zdroje pro referenci služby v kódu klienta.

```
public class Client
{
      @WebServiceRef(wsdlLocation="http://localhost:8080/service?
      wsdl")
      static Service service;

      public static void main(String[] args)
      {
            try { (new Client).doTest(args); }
            catch(Exception e) { e.printStackTrace(); }
      }

      public void doTest(String[] args)
      {
            try
            {
                  Port port = service.getPort();
            }
            catch(Exception e) { e.printStackTrace(); }
}     }
```

Vidíme tedy, že klient si pomocí anotace zjistí cestu k WSDL souboru, taky si získá port služby a může ze služby využít metody a operace, které ona nabízí. Ke zjednodušení tedy výrazně došlo i v tomto případě. Shrňme si postup v těchto krocích: [14]

1. tvorba kódu tříd klientskou webovou službou
2. použití nástroje s WSDL pro generování zdrojů
3. kompilace tříd klientskou webovou službou
4. spuštění klienta webové služby

Celý proces i zde můžeme usnadnit jakýmkoliv vývojovým prostředím (IDE)

3.2.4 Zhodnocení JAX-WS

Standard JAX-WS můžeme označit za velmi účelného a potřebného následníka JAX-RPC. Je zřejmé, že jeho výhoda není pouze v anotacích a přechodu na vyšší verze jazyka Java. JAX-WS přináší celou řadu nových nápadů a hlavně mnohem dokonalejší provedení v definování vazby Java typů na XML díky JAXB. Další výhody oproti JAX-RPC jsou uvedeny v předcházející podkapitole, kde jsme shrnuli všechny novinky. Většina z nich by se dala považovat za plusy a klady. Po přečtení této části tedy již víme, jak prakticky služby v jazyce Java vytvářet, jaké máme možnosti a omezení. Nyní bychom se podívali na obecnější náhled na vývoj služeb pomocí konzole a možnosti hostování takovýchto služeb.

4 Hostování WS v jazyce Java

V předchozích kapitolách jsme si popsali, jak taková skladba webových služeb vypadá a také to, jaké možnosti implementace a vývoje máme v jazyce Java. Krok po kroku jsme probrali návod jak vyvíjet WS pomocí JAX-WS. Touto metodou se budeme již jen nadále zabývat, jelikož JAX-RPC se v dnešní době v jazyce Java téměř nepoužívá. K vyzkoušení jednoduchých příkladů bylo použito vývojové prostředí Netbeans. Pro nás vývojáře má určitě své výhody implementovat WS ve vývojovém prostředí. Poskytne nám všechny námi chtěné knihovny, některé parametry WS u implementace lze jednoduše „*naklikat*“, nachystá nám svůj server, na kterém můžeme služby a program testovat a vygeneruje všechny potřebné soubory, jako jsou SOAP či WSDL. Zjednodušeně řečeno, ulehčí nám maximálně naší práci a my se nemusíme o tyto záležitostí starat. Zde ale přichází zásadní problém.

V reálné situaci bychom jistě s tímto přístupem moc nepochodili. Představme si situaci, že chceme službu nasadit do provozu zákazníkovi a vzali sebou projekt vytvořený například v našem vývojovém prostředí Netbeans. Úspěch této práce závisí na podmínkách u zákazníka. Nemusel by mít nainstalováno vývojové prostředí, mohl by využívat úplně jiný server, než na kterém jsme službu testovali, anebo na klientských počítačích je nainstalován úplně jiný operační systém. Je proto zavádějící se zamyslet nad tím, jak vyřešit tyto komplikované situace.

Jazyk Java nabízí vlastnosti, které mohou v těchto situacích pomoci. Jednak je to multiplatformní jazyk, tudíž její aplikace mohou běžet jak na operačních systémech Windows tak třeba Linux. Což nám jistě velmi pomůže, protože obecně webové služby jsou více využívány pro OS Linux. Další vlastností jazyka je ta, že samotný balíček Java jakékoliv verze nám nabízí příkazy, které může vývojář spouštět přes konzoli. Tím je zajištěno to, že nasazení služby bude možné na jakémkoliv počítači a za jakékoliv situace (každý operační systém konzoli obsahuje).

Proto se v této kapitole bude hlavně zabývat možnostmi vývoje a nasazení služby v jazyce Java pomocí příkazového řádku. Další částí pak bude najít jakýsi efektivní a nejlepší způsob hostování, aniž bychom využili jakékoliv vývojové prostředí. Hotová služba a její části tedy budou nachystány obecně, aby se daly nasadit a využít v jakémkoliv klientském prostředí a podmínkách.

Předpokládáme, že v místě, kde vyvíjíme tyto služby, jsou nainstalovány aktuální baličky, které Java na svých stránkách nabízí. V dnešní době verze Javy je již 1.7 a poskytuje nám dávkové *bat* soubory, které jako vývojáři využijeme pří implementaci služeb pomocí příkazového řádku. Druhým aspektem, který je nezbytný pro vývoj služeb, jako takových je nějaký webový aplikační server. Na výběr máme ty nejrozšířenější volně dostupné, jako jsou *Apache Tomcat*, *GlassFish* a nebo samotná Java nabízí svůj server *System Application Server Platform Edition*. Místo aplikačního serveru lze jako náhradu využít virtuální stroj Javy, ten

také probereme v této kapitole. Poslední možností by jistě bylo nasadit službu jako službu operačního systému. Zkusíme si vytvořit a pomocí příkazových řádků nasadit a spustit nějakou vzorovou webovou službu.

4.1 Vývoj služeb pomocí konzole a hostování na aplikační server

V několika krocích se budeme snažit dospět k tomu, abychom službu a následného klienta vytvořili a zprovoznili pomocí příkazové řádky. Jak již je zmíněno výše, k tomuto ukázkovému příkladu budeme mít nainstalován server (využijeme v této práci GlassFish). Pro zjednodušení práce je vhodné nastavit všechny instalační adresáře do proměnné PATH, abych bylo možné příkazy používat kdekoliv v systému. [15]

Implementace koncového bodu webové služby

V prvním kroku je třeba naimplementovat v jazyce Java samotnou webovou službu a její koncový bod. Jedná se o část ukázky, kterou již nebudeme vysvětlovat (viz předchozí kapitola).

```
@WebService()
public class Calculator
{
   @WebMethod(operationName="add", action="urn:Add")
   public int add(int i, int j)
   {
      int k = i +j;
      System.out.println(i + "+" + j +" = " + k);
      return k;
   }
}
```

Kompilace koncového bodu webové služby

Musíme mít spuštěn testovací webový server (GlassFish a příkaz *asadmin start-domain*). Potom je na řadě samotná kompilace kódu a vytvoření *class* souboru příkazem *javac.*

Vygenerování popisu služby WSDL

K samotnému vygenerování použijeme příkaz *wsgen*. Jeho dávkový soubor najdeme jak v složce Javy, tak v složce aplikačního serveru. V adresáři *endpoint* se vytvořila složka *jaxws*, která obsahuje jednotlivé části, které daná služba využívá. Tyto třídy jsou použity při provádění služby na aplikačním serveru. Dále se nám vytvořila sama složka *generated*, ve které už je samotný popis služby. Soubor *CalculatorService_schema1.xsd* definuje schéma služby pro soubor *CalculatorService.wsdl.*

Zabalení do WAR souboru a následné nasazení

Poslední krok, který se týká samotné služby a jejího následnému nasazení. K tomu je třeba zadat informace o službě. Webové služby dodávané jako servlety jsou zabaleny do WAR balíčku. Toho docílíme příkazem *asant* (příkaz poskytuje aplikační server). Pokud chceme znát strukturu WAR souboru, můžeme se podívat do souboru *build.xml*. Příkazy jsou následující:

asant create-war – vytvoří příslušný WAR soubor
asant deploy – nasadí službu na běžící aplikační server

Vytvoření a kompilace klienta k službě

Následuje implementace klienta, který využije nachystanou službu. Opět to budeme provádět pomocí příkazové řádky a zajistíme tím tak jakousi obecnost ve vývoji. Část zdrojového kódu je uvedena v kapitole 3.2.3. K popisu ukázky dodáme snad jen to, že klient získá koncový bod služby pomocí metody *getWebServiceRefNamePort*, kde *WebServiceRefName* je hodnota portu ve vygenerovaném WSDL souboru. Poté co klient získá koncový bod, může zavolat jednotlivé operace služby.

Import a vygenerování potřebných souborů

Zřejmě nejdůležitější krok před samotným nasazením klienta, kterému potřebujeme naimportovat soubory z dané služby. K tomuto se používá příkaz aplikačního serveru *wsimport*, který do zadané složky naimportuje soubory z URL adresy, na které je již naše služba spuštěna.

Spuštění klienta

K samotnému spuštění nám aplikační server poskytuje příkaz *appclient -mainclass client.JAXWSClient.* Na výstupu bychom pak měli objevit očekáváné operace klienta.

Zhodnocení a závěr

Tímto jednoduchým příkladem jsme si ukázali jak nachystat a spustit webovou službu a klienta jen pomocí příkazové řádky. Určitě složitější postup než ve vývojovém prostředí, ale za to obecnější a bude fungovat v různých podmínkách při instalaci. Nasazení jakýchkoliv služeb na aplikační server je tou nejpoužívanější metodou implementace. Postup při hostování je zcela jasný a jakékoliv vývojové prostředí si s tím dokáže poradit. My se k tomu všemu zabývali hostováním přes příkazový řádek, který svou obecností je také velmi potřebný.

4.2 Vývoj služeb pomocí konzole a hostování na virtuální stroj Javy

Druhá možnost nasazení služby spočívá v tom, že nám Java poskytuje svůj virtuální stroj – Java Virtual Machine (JVM) Ten můžeme nezávisle na platformě využít k nasazení a provozu služby. Tato možnost nám jakoby nahrazuje aplikační server, dovoluje nám službu zprovoznit a využívat a dokáže vygenerovat a zobrazit popis služby WSDL. Přičemž celý proces vývoje je totožný z předchozího příkladu, jen využijeme příkazy samotné Javy a v zdrojovém kódu dopíšeme metody pro nasazení virtuálního stroje.

Již nebudeme krok po kroku přesně popisovat nový celý příklad, ale zaměříme se na odlišnosti oproti předchozí metodě. K tomu bychom jako vzorový příklad mohli použít službu, která by zpracovala objednávku zboží a počet objednaných kusů, které by zákazník (v našem případě by se jednalo o klienta) chtěl a na tomto základě by vygenerovala zákaznické číslo, pokud by vše bylo zpracováno v pořádku. Výhodou této metody je, že pokud je služba naimplementována, nepotřebujeme již generovat další soubory či WAR baličky potřebné k nasazení služby. Jen se vytvoří třídy, které zajistí převod objektů Javy do XML a samozřejmě se vygeneruje WSDL a XSD dle rozhraní služby.

Nutno podoktnout, že způsob využití virtuálního stroje Javy jako možnost publikování služeb, je spíše za účelem otestování funkcionality. Je ale dobré tuto možnost zmínit a uvědomit si, jak funguje komunikace služby a klienta tímto jednoduchým API voláním.

Postup vývoje webové služby

Jak jsme již zvyklí, je třeba naimplementovat koncový bod služby pomocí našich známých anotací *@WebService* a *@WebMethod*. Dalším krokem je vygenerování pomocných a popisných souborů WSDL. Zase se nic neliší a použijeme příkaz *wsgen*. Až do této chvíle probíhá práce na vývoji služby stejně, tedy krom toho, že nezmiňujeme nic o aplikačním serveru, který nebudeme vůbec potřebovat.

Nyní musíme naimplementovat třídu, která nám zajistí zobrazení služby v prohlížeči. Nazveme tedy tuto třídu *Publisher*. Nahradí nám aplikační server a testovaní prostředí, ale použitelnost této funkce je jen v době, kdy je třída spuštěna a je zavolána třída *main*. Do jisté míry může být toto omezení docela limitující, na druhou stranu je služba nasazena a spuštěna a není zde žádné časové omezení. Po spuštění této třídy v konzoli stačí zadat do prohlížeče URL adresu, kterou nastavíme v samotné třídě, zpravidla *http://localhost:8080/Publisher*. Na adrese ověříme nasazení služby. Také si můžeme prohlédnout WSDL kód na adrese *http://localhost:8080/Publisher?wsdl*. Zdrojový kód třídy *Publisher* vypadá následovně.[16]

```
public class Publisher
{
      public static void main(String[] args)
      {
```

```
            Endpoint.publish("http://localhost:8080/Publishe
            r", new MyService());
        System.out.println("Sluzba je nasazena na
        http://localhost:8080/Publisher")
    }
}
```

Metoda *Endpoint.publish* nám poskytuje jednoduchý způsob, jak publikovat a testovat JAX-WS služby. Má dva parametry, umístění webové služby a samotnou třídu s implementací webové služby. Vytváří nám jednoduchý server na zadané URL a poskytne nasazení této služby do provozu. Server běží na Java Virtual Machine a může být ukončen metodou *Endpoint.stop*.

Postup vývoje klienta webové služby

Základ tvoří vygenerování klientských pomocných souborů z WSDL dané služby. JAX-WS využívá příkaz *wsimport*. Klient pak tyto soubory používá k zavolání webové služby, aniž by řešili formát SOAP zpráv. Dále je nezbytné, aby byla nasazena a spuštěna služba v třídě *Publisher* na virtuálním stroji Javy, jak je popsáno v předchozí podkapitole. Hlavní prvky zdrojového kódu vypadají následovně:

```
public class Client {
  ...
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out
          .println("Zadej URL adresu webove sluzby");
      System.exit(-1);
    }
    URL url = getWSDLURL(args[0]);
    ...
  }
  private static URL getWSDLURL(String urlStr) {
    URL url = null;
    try {
      url = new URL(urlStr);
    } catch (MalformedURLException e) {
      e.printStackTrace();
      throw new RuntimeException(e);
    }
    return url; }...
```

Metoda *getWSDLURL* najde adresu na virtuálním stroji Javy a předá ji klientské spouštěcí metodě *main*. Ta si tuto adresu převezme a dle funkce klienta ji dále využívá. Kód zde samozřejmě není celý, funkčnost a logika se doimplementuje dle toho, co dále má klient dělat.

Důležité je si uvědomit, jak se klient k souboru WSDL dostane. Posledním krokem je kompilace a spuštění klienta. To se opět provede pomocí konzole. Na výstupu bude ukazovat příslušná operace klienta a na výstupu spuštěné webové služby na virtuálním stroji vidíme reakce služby na operaci klienta. [16]

Zhodnocení a závěr

Tato možnost a metoda hostování nemusí být pro nás zajisté tak známá. Jak jsme již zmiňovali, použijeme ji vlastně jen pro testovací účely. Dopsáním pár drobností v kódu si službu a její funkčnost ověříme a můžeme ji posléze nasadit třeba na aplikační server. Jistě ale má v této práci své místo a proto ji zmiňujeme.

4.3 Vývoj služeb pomocí konzole a hostování v operačním systému

V této části bychom chtěli prozkoumat možnosti v programovacím jazyku Java, zda umožňuje, či poskytuje nástroje, pomocí kterých lze hostovat služby jako služby operačního systému. Budeme se snažit najít postup, kdy daná služba poběží v operačním systému a my ji můžeme zastavit, spustit, či automaticky zprovoznit dle potřebné situace. Samotná Java nám neposkytuje žádné API, pomocí kterého bychom jednoduše nastavili službu na hostování v operačním systému, což určitě přináší komplikace. Existuje ale možnost využít aplikace Java Service Wrapper. Ta samotná je naimplementována v Javě a dokáže aplikace pomocí *jar* souboru nasadit do prostředí operačních systémů jako službu. Samotné nastavení opět probíhá přes konzoli pomocí specifických příkazů této aplikace. Wrapper se vztahuje na všechny druhy implementací v Javě. My využijeme jen nasazení webových služeb do operačních systémů. Proto tedy nasadíme aplikaci do systému Windows, na kterém vyvíjíme programy pro celou tuto diplomovou práci.

Úvod do Java Service Wrapper

Samotný produkt je vyvíjen firmou Tanuki Software a je dostupný na adrese [17]. Zde samozřejmě nalezneme informace o produktu, dokumentace a stáhnutí jednotlivých verzi. Wrapper je dostupný na všechny druhy operačních systémů, na které dokáže nasadit Java aplikace jako služby těchto systémů. Vyřeší za nás mnoho problémů. Pokud se o webovou službu nechceme příliš starat, tak to za nás dělá operační systém. Wrapper poskytuje mnoho funkcí. Samotní vývojáři spolupracují s komunitou obyčejných uživatelů a konzultují společné kroky a problémy, které by mohl produkt vyřešit. My se zaměříme na hlavní vlastnost, o kterou se opřeme v této práci, a tou je nasadit a spustit webovou službu jako windowsovskou službu.

Přípravy na provoz

Abychom aplikace uvedli do provozu, musíme umět tento nástroj dokonale používat. K dispozici máme možnosti na různé systémy a také to, zda se jedná o 32 či 64 bitovou verzi. Ještě podotkněme, že Wrapper je pro obyčejné testovací možnosti, jaké v této práci provádíme,

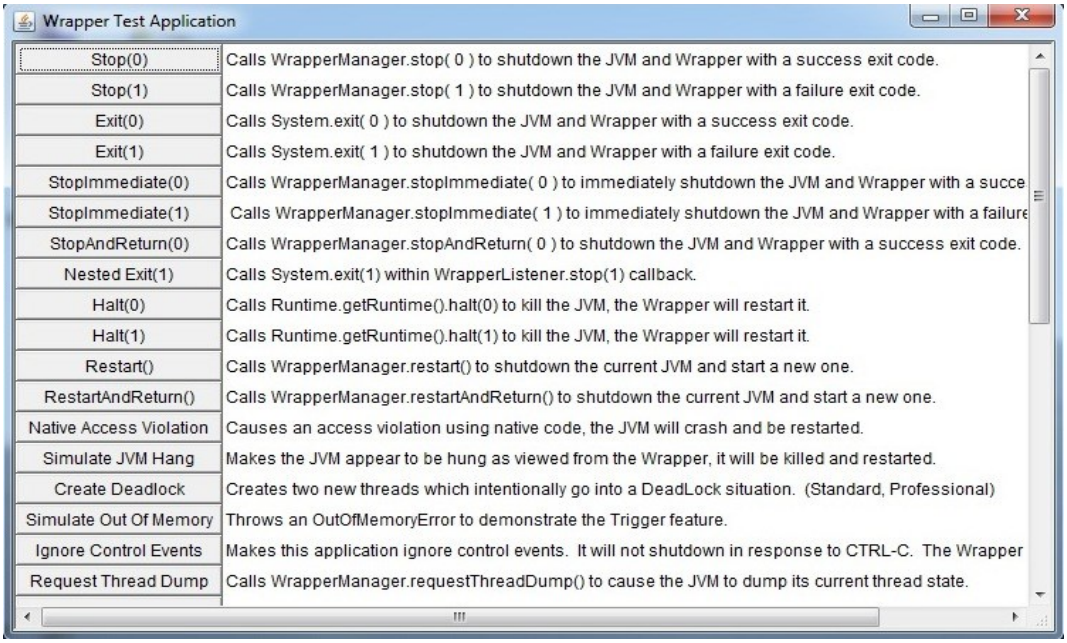
k dispozici zdarma a bez omezení hlavních funkčností. Instalace je jednoduchá, stačí stáhnutý instalační soubor rozbalit kamkoliv do systému. Adresářová struktura Java Service Wrapperu je následující:

- /bin – obsahuje spouštěcí a dávkové soubory, které využijeme v příkazovém řádku. Pro usnadnění práce vložíme tuto složku do proměnné PATH.

- /conf – obsahuje konfigurační soubory, které se starají o nasazení Java aplikace do operačního systému. Každá aplikace musí mít svůj vlastní konfigurační soubor, ve kterém je nastavena a následně spuštěna. Pro modifikaci nás hlavně zajímá soubor *wrapper.conf*, jenž si v návodu níže popíšeme důkladněji, co kde nastavit a dopsat.

- /lib – složka obsahuje spustitelné soubory jednotlivých aplikací a samotného produktu. Všechny aplikace, které chceme pomocí Wraperru nasadit, zde musí mít spustitelný soubor *.jar*. Cestu k němu specifikujeme v konfiguračním souboru.

- /doc – obsahuje dokumentaci a informace o licenci. [18]



Obrázek 7: Java Service Wrapper

Vytvoření služby v operačním systému Windows

První věc, kterou musíme udělat, je aplikaci naimplementovat, dle postupů výše uvedených kapitol. Proto je na místě využít aplikaci z předchozího příkladu, kdy jsme se bavili o hostování služeb na virtuálním stroji Javy. Vytvořený spustitelný soubor této aplikace *WS_Client_Server.jar* zkopírujeme do instalační složky Wrapperu do adresáře */lib*. Další věcí je propojit naší implementaci s programem Wrapper. Tento krok se odehrává pouze v nastavení konfiguračního souboru pro příslušné aplikace v adresáři */conf*. V něm nalezneme soubor *wrapper.conf*, který budeme upravovat dle následujících bodů.

- Změníme řádek, který je primárně nastaven na spuštění tříd programu Wrapper *wrapper.java.mainclass=org.tanukisoftware.wrapper.test.Main* na řádek *wrapper.java.mainclass=org.tanukisoftware.wrapper.WrapperSimpleApp*.
  Tím překonfigurujeme spuštění programu do testovacího prostředí, kde nasadíme naše aplikace.

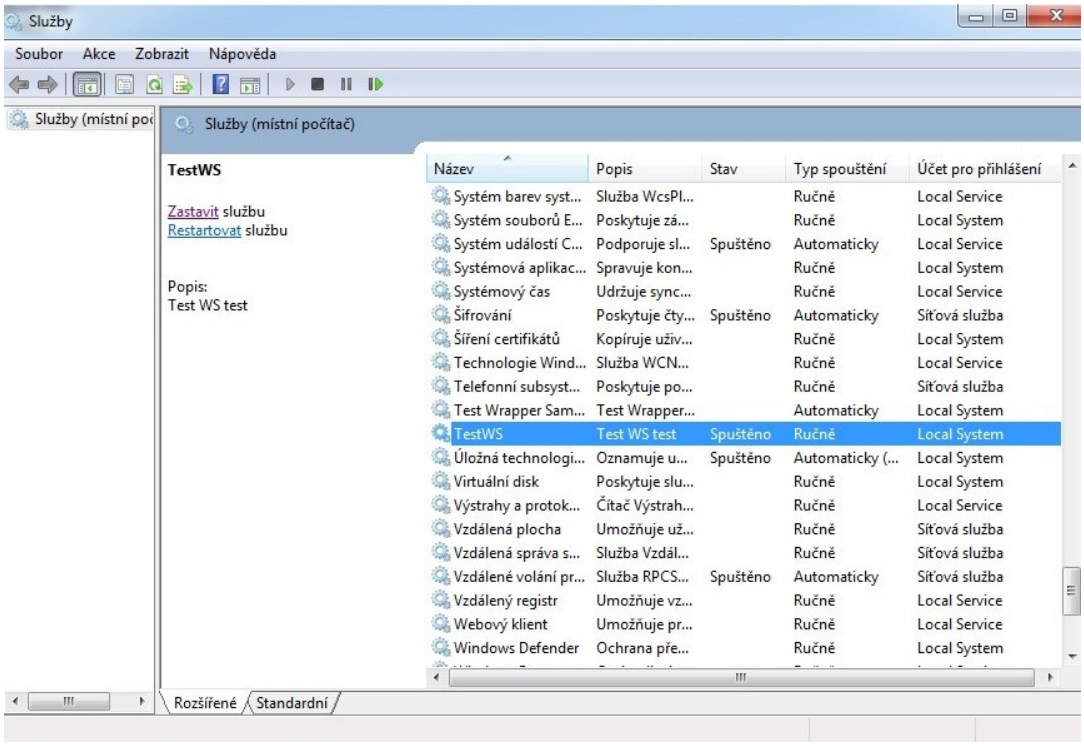
- Kousek pod tímto kódem najdeme část, kam definujeme spustitelné soubory *jar*. Vidíme zde nastavení na spuštění samotného programu Wrapper. Přidejme tedy tento řádek: *wrapper.java.classpath.3=../lib/WS_Client_Server.jar*
  Nastavili jsme jednoduše cestu, kde máme již uložen spustitelný soubor naší služby.

- Posuneme se dále a upravíme v souboru cestu k naší *main* třídě služby. Změníme řádek: *#wrapper.app.parameter.1=* na *wrapper.app.parameter.1=publish.Publisher*. Tady je důležité nezapomenout smazat znak #, jelikož ten značí komentář, tedy kód, který se při překladu přeskakuje. Tak by naše služba neměla žádnou metodu a parametr.

- Poslední částí konfigurace jsou již jen popisy a pojmenování služby, které pak uvidíme ve správě služeb v systému Windows. Na samotném konci konfiguračního souboru nastavíme příslušné řádky.
  wrapper.name=TestWS – jméno naší služby
  wrapper.displayname=TestWS – jméno naší služby, které se zobrazí ve správě služeb
  wrapper.description=Test WS test – popis služby [18]

To je ze strany konfigurace pro tuto chvíli vše. Poslední věci je jen převod služby do prostředí operačního systému, který nám Wrapper umožňuje.

Využijeme konzolový příkaze *InstallTestWrapper-NT*. Produkt projde konfigurační soubor a oznámí nám, že naše *TestWS* služba je nainstalována. O tom se snadno přesvědčíme, pokud si v systému Windows spustíme správu služeb a jejich seznam. Jak vidíme na obrázku 8, opravdu nám nástroj nainstaloval naší Java službu na Windows službu a můžeme ji lehce spustit. Případné zastavení jak v samotné správě služeb nebo opět pomocí konzole příkazy *StartTestWrapper-NT*, *StopTestWrapper-NT*. Opačným příkazem je pak *UninstallTestWrapper-NT*. Ten nám logicky službu ze systému Windows odstraní.



Obrázek 8: Hostování služby v OS Windows

Zhodnocení a závěr

Vzhledem k tomu, že Java neposkytuje žádné API jak převést tyto naše Java služby jako služby operačního systému, je Java Service Wrapper vynikající nástroj. Jak jsme si mohli vyzkoušet, tak jednoduchým nastavením v konfiguračním souboru dokáže převést jakoukoliv Java aplikaci do prostředí služeb ve Windows. Na druhou stranu najdeme i mínusy, které tento produkt má. Dokáže převést pouze aplikace, které poskytují *jar* soubory. V předchozích kapitolách jsme vyvíjeli i služby jako webové aplikace, kde je třeba vygenerovat *war* soubor. Tenhle bohužel Wrapper nasadit a hostovat nedokáže. Další nevýhodou může být, že umí nasadit a spustit jen jednu službu v jednu chvíli. Každá služba či aplikace využívá právě jeden konfigurační soubor. Samotná kostra nástroje je propojena soubory vedoucími k právě jednomu konfiguračnímu souboru, tudíž s tímto nic nenaděláme. Řešením by pak mohlo být hostovat tyto služby odděleně ve více Wrapper nástrojích nebo je spouštět postupně za sebou obměnou konfiguračního souboru. Samotný nástroj se neinstaluje, jen se jeho celá složka umístí nebo překopíruje kdekoliv do systému k aplikaci. Velikost nedosahuje ani 3MB, takže aby měla aplikace svůj zvláštní Wrapper, nemělo by to být pro nás žádným problémem.

5 Realizace šablonové služby

V této praktické kapitole si popíšeme, jakým způsobem a jakými jednotlivými programátorskými kroky jsme postupovali, aby ve výsledku vznikla služba, která bude jakousi šablonou pro jakékoliv reálné nasazení a využití. Po dohodě s vedoucím práce jsme se zaměřili na služby hostované v operačním systému (v této práci se jedná o Windows 7) hlavně z důvodu nedostatečného probádání a poznání této oblasti. Kapitola bude rozčleněna na postupné popisy jednotlivých provedených úkolů a částí aplikace s tím, že celá implementace je napsána v jazyce Java a v prostředí Netbeans. Nejprve se zaměříme na konkrétnější implementaci, která postupně bude přecházet do obecnější šablonové podoby. Jak již bylo zmíněno v předchozích částech, zdrojové kódy jednotlivých programů budou přiloženy v příloze.

5.1 Komunikace se službou hostovanou v OS Windows

V kapitole o hostování služeb jsme se zabývali pouze tím, jak nasadit do prostředí operačního systému danou službu, jak ji spustit a případně zastavit. To nám zajisté nemůže stačit. Nyní chceme se službou komunikovat a to nejlépe pomocí SOAP zpráv. Proto prvním úkolem musíme zjistit, jakým způsobem se posílají a přijímají zprávy v případě hostování služby v systému Windows. Zdrojový kód naší nasazené služby je nutné poupravit, aby služba přijímala zprávu v požadavku a vracela zprávu v odpovědi. K tomuto úkolu budeme používat formát XML zpráv, které jsou základním kamenem celé komunikace.

Popis problému

První zadání našeho problému vypadá tedy takto. Je nutné vytvořit dvě Java aplikace. První je webová služba hostovaná ve službách prostředí OS Windows. Druhá aplikace je konzolová aplikace, která ze souboru načte textový řetězec (ten bude obsahovat XML). Druhá aplikace se připojí na službu a naváže komunikaci tím, že jí v požadavku zašle načtený textový řetězec. Windows služba tento řetězec přijme a zašle v odpovědi zprávu konzolové aplikaci. Tato zpráva bude obsahovat nějaký jiný textový řetězec (opět ve formátu XML). Celá komunikace skončí tím, že konzolová aplikace tuto odpověď přijme a zapíše daný řetězec do jiného souboru. Nyní se budeme snažit tento základní proces realizovat a na něm budeme stavět další vylepšení a možnosti využití.

Implementace a vyřešení problému

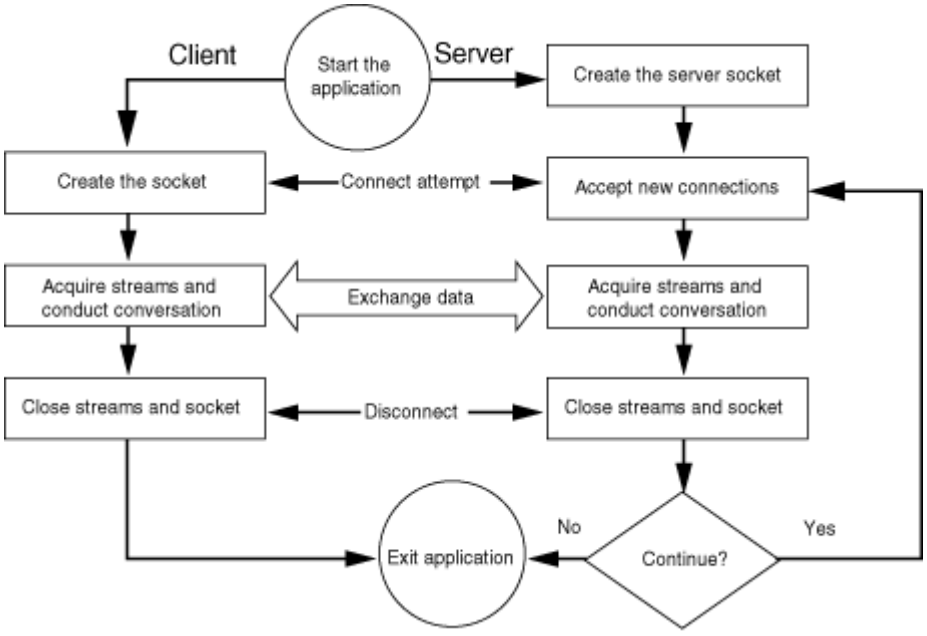
Jak je zřejmé, největším problémem je určitě navázání komunikace mezi aplikací a službou. Nebudeme zde uvádět zdrojové kódy, jen si popíšeme, jakými prostředky jsme toho dosáhli. Nejprve se zaměříme na Windows službu. Ta je nasazena do prostředí operačního systému stejným způsobem, jaký je popsán v kapitole 4.3. Princip propojení obou aplikací je v tom, že se chceme napojit na určitý port, na kterém služba je nasazena. V tomhle případě je

hostování ale umožněno pomocí Java Service Wrapperu, který si již port obsadil pro svůj běh. Proto realizujeme tuto komunikaci využitím možností jazyka Java a jeho knihovny *Socket* rozhraní. *Socket* je koncový bod pro komunikaci mezi dvěma aplikacemi a funguje v principu tak, že se aplikace domluví, který port si rezervují a tam na sebe čekají nebo provádějí komunikaci. Tento baliček nám pak umožňuje řadu metod pro bezproblémový přenos, v našem případě řetězce s XML. Windowsovská služba je postavena na tomto řádku kódu, kde stačí nastavit číslo portu, na který se má konzolová aplikace připojit.

```
ServerSocket providerSocket = new ServerSocket(port);
Socket connection = providerSocket.accept();
```

Konzolová aplikace vytvořená v Javě musí být v závislosti, o kterou službu se uchází, implementována velmi podobně. I ona využívá rozhraní *Socket* a musí být nastavena tak, aby se číslo portu shodovalo s číslem portu, na kterém naše služba čeká na komunikaci. Samozřejmostí je pak v parametru nastavit *hostname*, v našem případě by to byl *localhost*. Pro lepší pochopení je celý princip komunikace znázorněn na obrázku 9 a uvádíme *Socket* kód konzolové aplikace.

```
Socket requestSocket = new Socket ("hostname", port);
```



Obrázek 9: Princip naší komunikace mezi službou a klientem, převzato z [21]

Shrnutí pro přehlednost

Tento proces tvoří základ celé budoucí implementace, proto si ho ještě jednou tedy shrňme do těchto jednotlivých bodů, neboť se již měnit nebude.

1. Služba je hostována v prostředí Windows pomocí Java Service Wrapper.

2. Je spuštěna a čeká na určitém portu na zahájení komunikace konzolovou aplikací.

3. Konzolová aplikace se na port připojí, naváže komunikaci a z načteného souboru pošle v požadavku textový řetězec ve formátu XML.

4. Služba požadavek zpracuje, dá konzolové aplikaci vědět, že řetězec přijala a jako odpověď pošle jiný načtený textovým řetězcem ve formátu XML ze souboru.

5. Aplikace odpověď zpracuje, dá službě vědět, že řetězec přijala a uloží ji do jiného souboru.

6. Celý proces komunikace je konzolovou aplikací ukončen.

7. Služba ale běží dál a čeká na další možnou komunikaci. Její proces je ukončen až zastavením provozu služby v prostředí OS.

5.2 XML Schéma a Castorové třídy v procesu komunikace

Základní skladbu a proces programu máme za sebou, ale pokud si uvědomíme celkovou funkci, jistě dojdeme k názoru, že skrývá mnohá omezení či problémy v případě posílaných textových řetězců. V praxi bychom museli při změně souboru měnit celé XML, což je určitě nekomfortní věc, každá změna by znamenala výměnu textu, a pokud by těchto změn bylo hodně, nebyli bychom moc efektivní. Naštěstí nám nejenom Java, ale i jiné programovací jazyky mohou tento problém pomoci vyřešit s pomocí XML schémat. Pojďme se tedy na ně blíže podívat.

Pokud jako programátoři známe jazyk XSD, máme velikou výhodu v tom, že dokážeme z libovolného XML souboru sestavit jeho schéma. Tím si zajistíme vazbu na případné změny, kdy nemusíme měnit celý XML dokument, ale stačí změnit schéma. Možná si říkáte, že pořád pouze měníme textový řetězec formátu XSD místo XML, ale kouzlo se skrývá v tom, že existují generátory, které nám mohou vygenerovat ze schématu příslušného XML takzvané *Castorové třídy*. Co si pod tímto pojmem představíme, k tomu se dostaneme, jen připomínáme, že schémata jsou stručně popsány v kapitole 2.2.4 a v případě zájmu se o nich více dozvíme zde [9] anebo přímo na webu konsorcia [19]. V další implementaci je budeme používat, proto je dobré o nich něco více vědět.

5.2.1 Castorové třídy

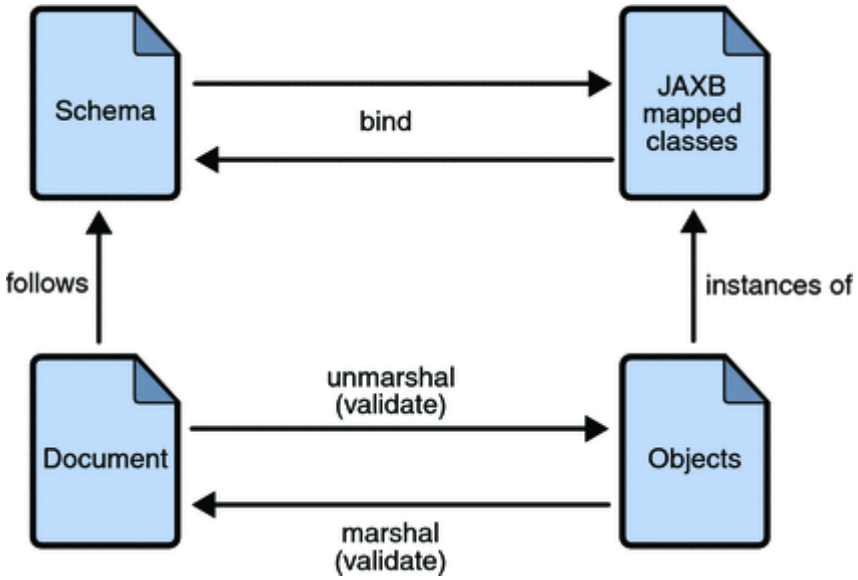
Jedná se o balíček tříd, které jsou vygenerovány z příslušného XSD souboru. Ten musí být validní vůči XML souboru, z kterého byl vytvořen, jinak se nám vytvoření těchto tříd nepodaří. Jednotlivé Castorové třídy odpovídají hlavním značkám (tágům) v XML dokumentu. Pokud jsou tyto značky rozšířeny o vlastnosti v podobě dalších značek, ty se promítnou

v třídách už jen jako atributy. Samotná skladba tříd je velmi pestrá, hlavně se jedná o *set/get* metody jednotlivých atributů a každé vygenerování obsahuje hlavní třídu *ObjectFactory.java*. Jak už název napovídá, pomocí této třídy jsme schopni vytvářet jednotlivé objekty a jejich instance, které máme v XSD, potažmo v XML definovány. Pomocí Castoru jsme schopni v jazyce Java naplnit instancemi a metodami příslušné data a využít vlastnosti jazyka, které nám umožní na tomto základě zpětně vygenerovat výsledný XML dokument.

Celý proces tedy funguje dynamicky a pomáhá nám při případných změnách nezasahovat přímo do XML souboru, ale pracovat s ním tímto způsobem. Pro nás programátory je určitě efektivnější a výhodné pozměnit jen XSD, z něho vygenerovat třídy, ty naplnit daty, které si přejeme, aby XML dokument obsahoval a na závěr vytvořit klasický XML soubor tak jak ho známe. Než se podíváme, jak vytvořit v Javě výsledné XML, něco málo k vygenerování ze schémat. Nástrojů je určitě mnoho, my využijeme klasický nástroj Javy, která nám nabízí příkaz *xjc.exe*. Struktura v příkazovém řádku vypadá takto: *xjc nazev_schematu.xsd*

5.2.2 Práce s Castorem v Javě

Po použití generátoru nám tedy vznikne balíček tříd odpovídající našemu XML schématu. Konečně tedy můžeme ve vývojovém prostředí tyto třídy naimportovat do projektu a efektivně s nimi pracovat. V principu budeme chtít naplnit Castorové třídy pomocí jejich instancí o data, které následně převedeme do XML souboru. Celý proces využijeme i naopak, z XML souboru získáme data, které uložíme do Castorových tříd a s nimi můžeme dále pracovat. Jak vidíme na obrázku níže, jazyk Java nám nabízí na tohle dvě knihovny *Marshaller* a *Unmarshaller*. [20]



Obrázek 10: Princip práce s Castorem v Javě, převzato z [12].

Knihovna Marshaller

Nejzákladnější operací v Castoru je vzít instance jednotlivých tříd a přiřadit je do XML formátu. Každá jednotlivá vlastnost je pak zastoupena v XML dokumentu. Takhle se sestavuje dané XML postupně do struktury, jak jí známe. Třídy samy o sobě neobsahují žádná data, pouze definují strukturu pro uložení dat. Data si v Javě nastavíme pomocí *set* metod jednotlivých objektu a instancí. Tyto metody jsou již generátorem vytvořeny automaticky. Teď už na ukázku jen uvedeme jakým kódem se tento převod z XSD do XML uskuteční.

```
Produkt p = new Produkt();
p.setNazev("HP ProBook");
JAXBContext obj = JAXBContext.newInstance(Produkt.class);
Marshaller marsObj = obj.createMarshaller();
marsObj.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marsObj.marshal(p, new FileOutputStream("C:/Produkt.xml"));
```

Pro názorný příklad je vytvořena instance třídy *Produkt* (jedná se o Castorovou třídu), která je metodou *setNazev* naplněna daty. Dále je implementován JAX objekt, na který je navázáno vytvoření třídy *Marshaller*. V posledním kroku je nastaveno formátování dle standardu XML a metodou *marshal* je celá instance uložena do XML souboru, který poté vypadá následovně.

```
<?xml version="1.0" encoding="UTF-8"?>
<produkt>
    <název>HP ProBook</název>
</produkt>
```

Knihovna Unmarshaller

Je nám jistě zřejmé, že tato knihovna funguje přesně naopak, než předchozí ukázka. Použijeme ji v situaci, kdy z XML souboru chceme využít data k další práci. K těmto datům přistupujeme pomocí vygenerovaných metod *get* a můžeme je pro další práci buď uložit do Castorových tříd, nebo rovnou je předávat jako parametr, popřípadě si je vypsat na konzoli. Rovnou tedy přejdeme k zdrojovému kódu, který nám data zpracuje.

```
Produkt p = new Produkt();
FileReader in = new FileReader("C:/Produkt.xml"));
JAXBContext obj = JAXBContext.newInstance(Produkt.class);
Unmarshaller unmarsObj = obj.createUnmarshaller();
p = (Produkt)unmarsObj.unmarshal(in);
System.out.println(p.getProdukt().getNazev());
```

Zdrojový kód je velmi podobný předchozímu. Opět si vytvoříme instanci třídy *Produkt* a načteme si XML soubor. Dále je implementován JAX objekt, na který je navázáno vytvoření třídy *Unmarshaller*. Do instance třídy *Produkt* jsou pomocí metody *unmarshal* předána veškerá data. K nim pak pomocí metod *getProdukt* a *getNazev* přistupujeme a můžeme si je např. pro kontrolu vypsat nebo s nimi pracovat dále.

Základní práci s Castorem máme za sebou a nyní si rozšíříme naší implementaci služby, kdy navážeme na předchozí komunikaci a naprogramujeme na ni tyto knihovny, čímž si zajistíme dynamickou komunikaci mezi aplikací a službou.

5.2.3 Implementace Castoru do služby

Dalším úkolem v naší implementaci je tedy rozšířit jak konzolovou aplikaci, tak i službu hostovanou v OS o možnosti XSD a Castoru. Důvod dynamického upravování pouze XSD jsme si již vysvětlili a komunikace je ve stejném principu, jak je popsána v kapitole 5.1. Proto jsme původní posílané XML, posílané službě i aplikaci, přepsali do formátu jejich schématu a vygenerovali jsme příkazem *xjc* Castorové třídy. Jak do aplikace, tak do služby jsme tyto třídy naimportovali a nyní jsme mohli nastavit příslušná data, které chceme v komunikaci posílat. Posledním krokem bylo dopsání zdrojového kódy výše popsané knihovny *Marshaller*. Zde jsme zatím nepoužili opačnou knihovnu Un*marshaller*, jelikož přenos zpráv nemá na sebe žádný vliv a obě aplikace posílají soubory nezávisle na jejich obsahu. Shrňme si tedy v bodech změny, které se liší od předchozí implementace.

1. Jsou vytvořeny XML schémata na základě představy, jakou strukturu XML chceme posílat v komunikaci mezi konzolovou aplikací a Windows službou.

2. Z těchto schémat jsou vygenerovány Castorové třídy. V případě změn se upraví již jen XSD a provede se nové vygenerování.

3. Windows služba i konzolová aplikace si pomocí vygenerovaných metod *set* nastaví přes instance Castorových tříd data pro komunikaci, která jsou následně převedena pomocí knihovny *Marshaller* do struktury XML souborů a jako soubor jsou uložena.

4. Proces pak pokračuje komunikací shrnutou na konci kapitoly 5.1.

5.3 Demonstrativní implementace klienta a služby

Technologie a principy, které jsme si v předcházejících částech této kapitoly popsali, využijeme k vytvoření ukázkové konkrétní situace, kdy budeme simulovat komunikaci služby a klienta na jednoduchých matematických operacích. Pro začátek bez použití SOA architektury. Vytvoříme tedy klienta, který službě pošle XML soubor v matematickém tvaru, ta ho zpracuje a vrátí nám výsledek přikladu. Tento příklad by měl sloužit pro následnou realizaci šablonové služby, která by se dala jednoduše použít pro vyřešení obdobných případů. V následujícím

popisu aplikace použijeme pojmy, které jsme vysvětlili již v dřívějších kapitolách. Proto již nebudeme teorii opakovat, jelikož by nám mělo být vše jasné.

5.3.1 Klientská část

Klient bude mít v tomhle konkrétním příkladu jednoduchou roli. Jen pošle dané matematické XML a počká si na výsledek, což bude práce služby hostované v operačním systému. Může se zdát, že to zní jednoduše, ale za prací klienta se skrývá mnohem více detailů. Musíme vymyslet, jaký tvar bude mít posílané XML, aby jej služba efektivně zpracovala. Tady použijeme myšlenku derivačního stromu. Soubor se dá rozložit na jednotlivé elementy a dle jejich druhu či atributu služba pozná, kterou operaci použít. Pokud si představíme zanořené matematické rovnice, právě derivační strom se nám bude velice hodit. Na základě příkladu sestavíme strom a projdeme ho od konce na začátek. Tím služba bude postupně vyhodnocovat rovnici a dojde k výsledku. Schéma posílaného XML souboru bude například vypadat následujícím způsobem:

```
<Node>
      <operace atr = "+">
            <hodnota>10</hodnota>
            <operace atr = "/">
                  <hodnota>6</hodnota>
                  <hodnota>3</hodnota>
            </operace>
      </operace>
</Node>
```

Tuto myšlenku tedy budeme realizovat. Napíšeme si příslušné XML schéma a vygenerují se Castorové třídy. V implementaci si naplníme tyto třídy daty, v našem případě libovolnými čísly a operacemi sčítání, odčítání, násobení a dělení. Právě atribut operace je v této části velmi důležitý. On bude rozhodovat, jak se bude služba chovat a kterou matematickou operaci musí provést. Pokud máme Castorové třídy nachystané, provedeme transformaci do XML, v Javě pomocí popsaných metod knihovny *Marshaller*. Komunikaci provádíme již dle známého *Socket* rozhraní a klient je připraven k použití. Jeho zdrojový kód naleznete v příloze k práci.

5.3.2 Hostovaná služba

Hlavním úkolem služby je přijmout XML dokument od klienta, analyzovat ho a správně vypočíst výsledek. Služba je neustále zapnutá, nahostovaná v OS a čeká na komunikaci přes *Socket* rozhraní. Implementace obsahuje stejné Castorové třídy jako v programu klienta. Ty zde jsou hlavně z hlediska pomoci, aby do nich služba přes knihovnu *UnMarshaller* uložila data z matematického XML dokumentu, které od klienta přijme. Tyto data posléze zpracovává, jak bylo řečeno, způsobem derivačního stromu, aby rovnice a výrazy v závorkách byly vyhodnoceny dle matematických pravidel. Všechny operace se řeší v této jediné službě pomocí

webových metod. Později, v myšlence SOA, si v této práci představíme efektivnější řešení daného problému. Následně je vytvořeno schéma XML souboru obsahující konečný výsledek. Z něho jsou vygenerovány jeho Castorové třídy a opět pomocí metody knihovny *Marshaller* vytvořen finální XML soubor určený pro poslání zpět klientovi.

5.4 Šablonová služba

V diplomové práci jsme došli k okamžiku, kdy je třeba zobecnit naše řešení tak, aby bylo možné jej použít na kteroukoliv situaci a taky co nejvíce ulehčit případným budoucím uživatelům práci. Celá aplikace by měla být efektivně navržena, aby splňovala všechny možnosti hostování služeb a aby její použití i pro nezainteresované uživatele bylo co nejjednodušší. Samozřejmě konkrétní implementace služby a na ni navazujícího klienta si musíme napsat dle našich požadavků sami. Než se pustíme do popisu jednotlivých řešení, shrneme si základní myšlenky, jak postupovat v návrhu šablonové služby. Na začátek si řekněme, že funkčnost šablonového řešení budeme zkoušet na již známém konkrétním řešení kalkulačky. Každou možnost hostování navrhneme efektivně jako sadu kroků, které jdou pěkně za sebou a odpovídají reálné situaci nasazení služeb do provozu. Pokud tyto kroky pro každou možnost spustíme, můžeme si je představit jako skript, který během okamžiku proběhne, a my máme službu nasazenou na konkrétní situaci. Ve výsledku vytvoříme jakési rozhraní, které bude tyto metody zahrnovat, a jednotlivé třídy pro možnosti hostovaní budou toto rozhraní implementovat. V rámci této kapitoly budeme řešit tyto situace:

1. Debug – základní nasazení služby do konzole přes *.jar* soubor a následné ladění či testování správné funkčnosti.

2. Windows služba – nasazení služby a jejího *.jar* souboru do prostředí operačního systému za pomocí Java Service Wrapperu.

3. Aplikační server – nasazení služby a jejího *.war* souboru na spuštěný webový server, v našem případě GlassFish.

V následujících podkapitolách si podrobně popíšeme tyto možnosti a to včetně návrhu a samotné implementace. O tu rozšíříme klienta našeho konkrétního příkladu, abychom mohli otestovat funkčnost celé šablony. Zavedeme nový balíček těchto šablonových tříd, který bude možné naimportovat do ostatních projektů.

5.4.1 Debug implementace

V první a nejjednodušší možnosti se zaměříme na debug mód. Jeho účel je základní nasazení služby a komunikace s ní přes konzoli. Zde můžeme danou službu efektivně otestovat, odladit a nachystat na další možnosti hostování. Návrh této situace je vcelku jasný. Daná služba, která plní nějakou funkčnost (v našem případě kalkulačka), při implementaci vygeneruje spustitelný jar soubor, který v příkazovém řádku můžeme spustit. Obyčejný uživatel tedy musí jen zadat správnou cestu k tomuto souboru a naše šablonová třída řešící tuto situaci se již

postará o zbytek práce. To znamená o nasazení, spuštění a otestování funkčnosti na straně klienta, který k službě přistoupí.

V tomto i dalších případech použijeme při implementaci v Javě třídy a jejich metody typu *Runtime* a *Process* k vytvoření prostředí zadávání příkazu tak, jak bychom měli klasický příkazový řádek. Program bude vkládat příkazy a splňovat jednotlivé kroky při nasazení. Program zadá příkaz do konzole pro spuštění .jar souboru a poté spustí klienta, který otestuje funkčnost služby v provozu. Celá komunikace si navzájem pro větší přehlednost vyměňuje zprávy o tom, v které fázi procesu se zrovna nachází. Celá tato šablonová třída implementuje proces v následujících krocích:

1. Uživatel musí implementovat službu a klienta, aby bylo možné šablonu použít.
2. Před spuštěním šablony uživatel pouze vyplní fyzickou cestu *.jar* souboru služby.
3. Pomocí příkazů do konzole je spuštěna služba z dodaného *.jar* souboru.
4. Spustí se dodaný klient, do kterého je šablona nasazena a který komunikuje s již spuštěnou službou na základě naimplementované funkční logiky.
5. Celý průběh procesu vidíme v konzoli a jsme informování o každém kroku.

5.4.2 Windows služba implementace

Druhou možností jsme se v této práci velmi důkladně zabývali. Jedná se o nasazení služby do prostředí operačního systému za pomocí nástroje Java Service Wrapper. Nebudeme již zde popisovat instalační postupy nástroje, které jsou zmíněny v kapitole 4.3, jen se podíváme na samotný návrh a implementaci této šablonové třídy. Před samotným použitím si ale samozřejmě musí uživatel tento nástroj nakonfigurovat na svou vlastní službu. V programu pak jen zadáme fyzickou cestu k *.bat* souboru Wrapperu, konkrétně k instalaci a nastartování. Šablona se pak postará o zbytek práce. Nasadí službu do prostředí OS, spustí ji a zároveň spustí i klienta, který může tuto službu využít ke svému účelu.

V této implementaci musíme vyřešit menší problém. Než je Wrapper schopen nasadit službu do prostředí Windows, potřebuje na to pár sekund času. Totéž platí, než se služba nastartuje. Skript při spuštění v žádném okamžiku na nic nečeká a může nastat chybová situace. Než se skutečně služba nasadí, už žádáme spuštění klienta a došlo by nám k selhání programu. V implementaci jsme tuto situaci vyřešili nastavením časového okamžiku přes cyklus, který jsme rovnoměrně rozdělili, aby tyto kroky byly schopny být vyřešeny včas, než se uskuteční další krok závislý na předchozím. Šablona tedy nainstaluje službu. Za krátký okamžik službu v OS spustí a pak nastartuje klienta, který se službou pracuje. Myslíme i na možnost celé odinstalace hostované služby, ale v rámci řešení je tato volba volitelná jen v případě nutné potřeby. Kroky této šablonové služby vypadají následovně:

1. Uživatel musí implementovat službu a klienta, aby bylo možné šablonu použít.

2. Uživatel musí službu nakonfigurovat v Java Service Wrapperu.

3. Před spuštěním šablony uživatel pouze vyplní fyzickou cestu k *.bat* souboru instalace a spuštění Java Service Wrapperu.

4. Pomocí příkazu do konzole aplikace nasadí službu do OS.

5. Spustí se dodaný klient, do kterého je šablona nasazena a který komunikuje s již spuštěnou službou na základě naimplementované funkční logiky.

6. Celý průběh procesu vidíme v konzoli a jsme informování o každém kroku.

5.4.3 Aplikační server implementace

Poslední možností jsme se dosud moc v této práci nezabývali, ale jedná o nejčastější a nejpoužívanější využití webových služeb v praxi. Proto tuhle možnost nasazení služby na aplikační server nesmíme opomenout. Existuje celá řada těchto serverů, v naší implementaci použijeme GlassFish, který je součástí většiny vývojových nástrojů pro Javu a splňuje všechny naše požadavky na provoz. Uživatel spustí server a zadá jen fyzickou cestu pro konkrétní konzolové příkazy nasazení služby daného serveru a cestu k samotnému *.war* souboru služby. Šablona se postará o zbytek práce. Zařídí nasazení služby přes konzoli, spustí ji a rovněž zprovozní klienta. Jako volitelnou možností jsou rovněž v implementaci nachystány příkazy pro vysazení služby z provozu serveru. Navíc kvůli přehlednosti otevře tato šablonová třída prohlížeč s adresou, kde právě daná služba je spuštěna. Jednotlivé kroky v této šabloně jsou následující:

1. Uživatel musí implementovat službu a klienta, aby bylo možné šablonu použít.

2. Je spuštěn aplikační server, kde je možné hostovat webové služby.

3. Před spuštěním šablony uživatel pouze vyplní fyzickou cestu k *.war* souboru služby, případně cestu k příkazům serveru pro nasazení služby.

4. Pomocí příkazu do konzole skript nasadí službu na aplikační server a spustí ji a otevře se prohlížeč s aktuální adresou spuštěné služby.

5. Spustí se dodaný klient, do kterého je šablona nasazena a který komunikuje s již spuštěnou službou na základě naimplementované funkční logiky.

6. Celý průběh procesu vidíme v konzoli a jsme informování o každém kroku.

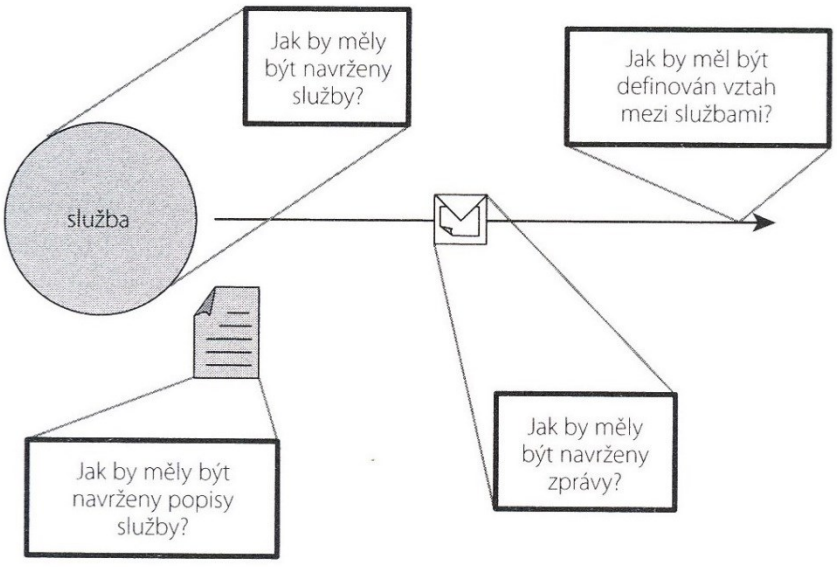
Ve všech případech jsme vytvořili základní šablony, které usnadní práci při nasazování služeb v různých možnostech hostování. Šablonový balíček obsahuje třídy, které po vyplnění základních údajů uživatelem zajistí pohodlné nasazení, spuštění a otestování funkčnosti služby. V této práci jsme toto obecné řešení vyzkoušeli na konkrétním demonstrativním příkladu implementace kalkulačky popsané v předchozích kapitolách.

6 Realizace SOA

V poslední kapitole této diplomové práce se zaměříme na servisně orientovanou architekturu. Princip této myšlenky vychází z toho, složit naší aplikaci ze skupinky nezávislých komponent, v našem případě webových služeb. To, jak v jazyce Java vytvářet služby, již známe. Teď jen zbývá si „pohrát“ s našimi vzorovými příklady a udělat z nich architekturu, která odpovídá metodologii SOA. Úvodem se podíváme na základní principy a pravidla SOA architektury, abychom s ní mohli účelně pracovat. V další části se zaměříme na demonstrativní implementaci, která bude vycházet z příkladu v kapitole 5.3. Na závěr se budeme snažit pomocí metod servisně orientované analýzy a návrhu naimplementovat praktičtější příklad SOA.

6.1 Principy a vlastnosti SOA

V této teoretické kapitole se seznámíme se sadou vlastností, pravidel a principů, které musí architektura splňovat, abychom ji mohli právoplatně nazývat servisně orientovanou. Současná SOA představuje architekturu, která podporuje servisní orientaci za použití webových služeb. Jednotlivé principy servisní orientace musí splňovat základní klíčové aspekty a týkají se již u části návrhu, jak ukazuje následující obrázek 11:[5]



Obrázek 11: Principy SOA se týkají problémů návrhů, převzato z [5]

- *Volná vazba* – služby udržují vztahy, které minimalizují závislosti a pouze vyžadují, aby si služby sebe vzájemně uvědomovaly.
- *Kontrakt služeb* – služby zachovávají dohodu na komunikaci, jak souhrnně definoval jeden nebo více popisů služeb a související dokumenty.

- *Samospráva* – služby mají kontrolu nad logikou, kterou zapouzdřují a nejsou závislé na jiných službách, aby mohly dokončit svou úlohu.
- *Abstrakce* – služby skrývají logiku před okolním světem, tedy kromě popisu dohody služeb.
- *Znovupoužitelnost* – logika je dělena do služeb za účelem podpory opětovného použití.
- *Kompozice* – kolekce služeb lze uspořádat a složit, aby vytvořily spojené služby.
- *Zjistitelnost* – služby jsou navrženy tak, aby byly navenek popisné, takže je lze vyhledat a zhodnotit dostupným vyhledávacím mechanismem.
- *Bezstavovost* – služby by měly být navrženy tak, aby byly maximálně bezstavové, bez ohledu na to, kam bude přesunuta správa informací o stavech.

Když už známe součásti tvořící základní architekturu a principy návrhu, které lze použít pro modelování a standardizaci těchto komponent, jediné co nám schází je implementační platforma, se kterou můžeme tyto části dát dohromady, abychom vytvářeli servisně orientovaná řešení. Sada již dříve zmíněných technologií webových služeb nabízí právě takovou platformu. Proto se tedy hlavní technologická sada XML stala běžnou součástí distribuované architektury internetu. Rovněž nám nyní poskytuje základní reprezentaci dat a vrstvu správy dat pro SOA. První generace webových služeb vzešla z vývoje tří klíčových standardů – WSDL, SOAP a UDDI. Zatímco UDDI je stále pomocný vyhledávací mechanismus pro většinu prostředí, WSDL a SOAP se staly hlavními technologiemi, které na standardu XML staví základní systém komunikace pro SOA. Proto jsme se v této práci s nimi tak důkladně seznámili.

6.2 SOA v jazyce Java

Veškeré principy, specifikace a vlastnosti, kterým se v této kapitole věnujeme, vyžadují samozřejmě podporu ze strany platformy, která nám dává dohromady funkční servisně orientovanou architekturu. Jazyk Java a její platforma *Java EE* je jednou ze dvou primárních technologií (další je *.NET*), které se v současné době pro vývoj SOA používají.

6.2.1 Podpora SOA

Obrázek 12 ukazuje podkladové vrstvy poskytované platformou *Java EE*, které podporují servisně orientovaná řešení. Platforma nám nabízí vývojové a běhové prostředí, jehož prostřednictvím lze realizovat tyto charakteristiky SOA následujícím způsobem: [5]

- *Zapouzdření služeb* – distribuovaná povaha nám umožňuje pomocí komponent vytvářet nezávislé jednotky procesní logiky. Ty mohou obsahovat menší či větší množství aplikační logiky a mohou být složeny tak, aby jednotlivé jednotky zahrnovaly

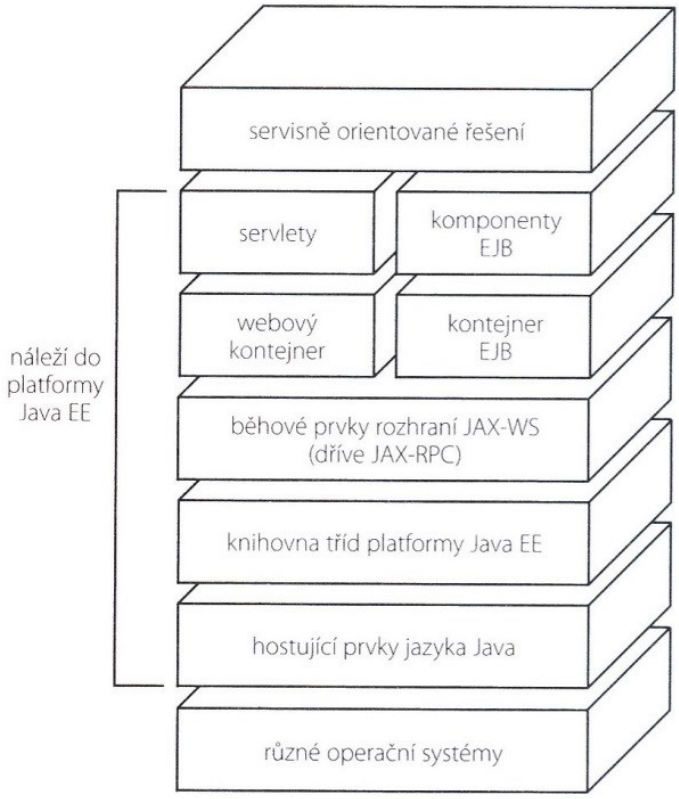
požadavky konkrétní úlohy nebo celého řešení. Pomocí webových služeb je lze zapouzdřit, čímž se z nich stanou koncové body služeb rozhraní JAX-WS.

- *Volná vazba* – použití rozhraní umožňuje odvozování metadat z vlastní logiky dané komponenty. Koncové body a rozhraní JAX-WS dále stanoví standartní WSDL definici podporovanou hostujícími službami. Volná vazba je tedy charakteristikou, kterou lze docílit pro podporu SOA.

- *Předávání zpráv* – pro podporu webových služeb nabízí aplikační rozhraní JAX-WS prostředky umožňující předávání SOAP zpráv přes protokol HTTP.

- *Rozšiřitelnost* – lze řešení navrhnout se službami, jež podporují představu budoucí rozšiřitelnosti. Ta se pak seskupuje na základě návrhové charakteristiky, která na úrovni rozhraní služby vnucuje určité konvence a jistou strukturu.

- *Abstrakce na úrovni logiky* – koncové body služeb rozhraní JAX-WS lze navrhnout do vrstev se službami, které odvozují aplikačně specifickou či znovupoužitelnou logiku.



Obrázek 12: Vrstvy platformy Java EE související se SOA, převzato z [5]

6.2.2 Podpora principů servisní orientace

Nyní se podíváme na čtyři principy servisní orientace, které nejsou přímo automaticky podporovány a poskytovány webovými službami. Stručně si probereme možnosti, jak je prostřednictvím platformy Java EE realizovat: [5]

- *Autonomie* – aby byla služba plně autonomní, musí být schopna nezávisle řídit zpracování své podkladové aplikační logiky. Pro koncové body služeb rozhraní JAX-WS je snazší udržet si autonomii zejména tehdy, když musejí provádět pouze malou část logiky.

- *Znovupoužitelnost* – komponenty společně se samotným programovacím jazykem Java podporují objektovou orientaci. To znamená, že znovupoužitelnost je dosažitelná na úrovni komponent. Na úrovni služeb sestupuje až k návrhu řídící logiky a koncového bodu služby.

- *Bezstavovost* – koncové body služeb rozhraní JAX-WS mohou být navrženy tak, aby existovaly jako bezstavové, avšak rozhraní JAX-WS poskytuje prostředky pro správu stavových informací pomocí objektu *HTTPSession*.

- *Zjistitelnost* – vyžaduje velmi dobře promyšlený návrh. Aby byla nějaká služba zjistitelná, je třeba věnovat pozornost návrhu koncového bodu, který musí být popsaný v maximální možné míře.

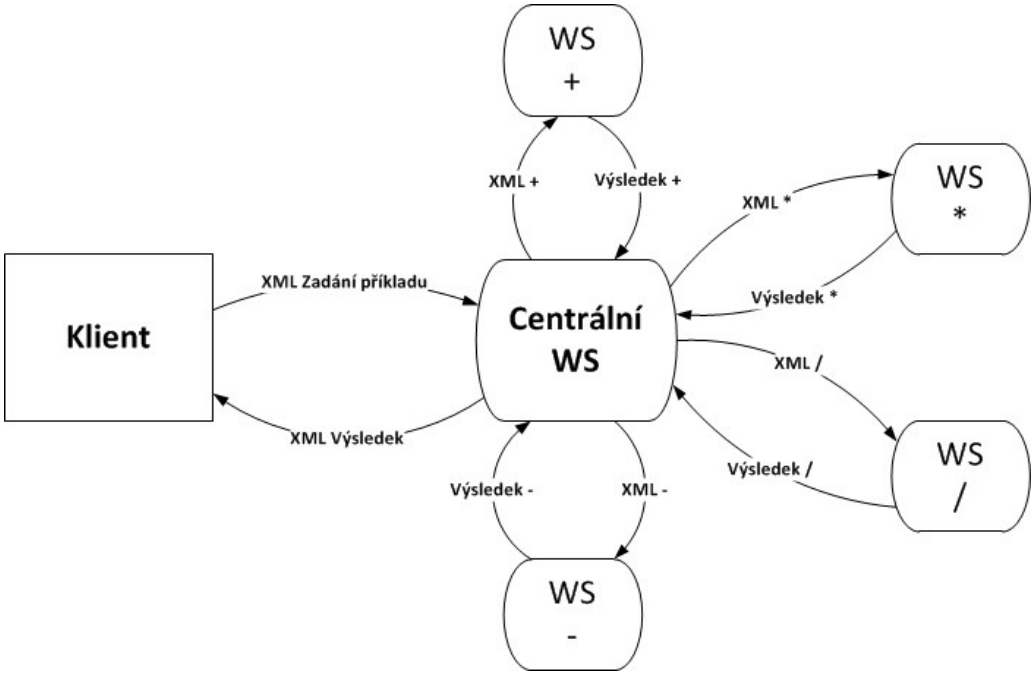
6.3 Rozšíření demonstrativní implementace o SOA

Tím, že jsme si již vytvořili aplikaci kalkulačky, kdy klient posílá službě XML zprávy v matematickém formátu, můžeme zkusit implementovat tento problém v rámci SOA a jejich pravidel. Hlavní myšlenka v podobě nezávislosti jednotlivých modulů (webových služeb) nás navádí k vytvoření vlastních služeb pro každou jednotlivou matematickou operaci. S nimi by komunikovala jedna centrální služba, která by od klienta dostala XML soubor se zadáním příkladu. Ta tedy analyzuje soubor a jednotlivé části dle potřeby zašle dalším službám, které vypočítají příslušné operace. Pro lepší přehlednost vidíme návrh SOA řešení na obrázku 13.

Z něho je tedy patrné, jak celý proces funguje. Důležité je si uvědomit, že všechny služby jsou hostovány v systému Windows a že všechny běží paralelně. Každá služba tedy po provedení příslušných operací opět čeká na případné nové spojení. Klient tedy může posílat více zadání, jelikož je celá architektura neustále připravena. Pokud by nastal případ selhání některé z operací WS, neovlivní to činnost jiných operací. Tím je tedy zaručena ona nezávislost celé aplikace.

Samotná implementace v jazyku Java se u klienta vůbec neliší, je tedy shodný s popisem v části 5.3.1. Jiné je to samozřejmě u webových služeb. Začněme nejprve tou centrální WS. V ní implementujeme Castorové třídy pro uložení dat od klienta a Castorové třídy

pro poslání výsledku, ale musí mít také Castorové třídy pro každou operaci. Ty slouží k uložení zpracovaných dat a jejich následné poslání příslušnému uzlu operace. Tím se tedy centrální služba stává jak službou, která přijímá zadání příkladu, tak i klientem, který rozesílá jednotlivé operace svým uzlům. To kde co jak má poslat pozná centrální služba pomocí atributu operace v samotné XML zprávě. Na základě jeho znaku jasně ví, kterou službu operace využije. Zavolá příslušnou třídu a ta se již chová jako klientská, která čeká na spojení s třídou operace. Přenos jako všude probíhá přes *Socket* rozhraní. Jak tedy vypadají uzly této architektury si popíšeme jen na jednom z nich. Pochopitelně princip zbylých je totožný, jen se mění matematická operace. Ty tedy obsahují Castorové třídy pro uložení dat na výpočet a Castorové třídy pro poslání dat s mezisoučtem. Ten je vypočítán pomocí webových metod a odeslán opět v podobě XML souboru na základě známých knihoven *Marshaller*. Celý proces skončí v případě, kdy je postupně zpracován celý dokument se zadáním, došli jsme k správnému výsledku a můžeme buď poslat zadání nové, nebo služby zastavit.



Obrázek 13: Návrh demonstrativní implementace SOA

V rámci rozšíření této práce se budeme u SOA implementací zabývat i řešením na platformě Java EE. Využijeme poznatky z řešení hostování služeb do prostředí operačního systému a implementujeme tento celek jako webovou aplikaci, která má na starosti jednotlivé webové služby v SOA architektuře. V praktickém použití také velmi častá možnost a v jazyku Java mnohem častější než hostování v prostředí OS. Klient zůstane nezměněn a princip matematických operací taktéž. Celou webovou aplikaci, která zahrnuje jednotlivé služby, jak je ukázáno na obrázku 13, naimplementujeme jako webový servlet. Ten tyto služby do sebe zabalí

a jejich funkcionalita, skladba a řešení je totožné s předchozím příkladem. Opět použijeme Castorové třídy a *Socket* rozhraní, ale jelikož v předchozím případě byly služby hostovány v OS a tam i paralelně spuštěny, teď musíme onen paralelismus naimplementovat. V jazyku Java se většinou řeší paralelní běh jednotlivých částí programů přes vlákna. Spuštění jednotlivých webových služeb v SOA architektuře naprogramujeme do vláken a tím si zajistíme souběžný běh celé architektury. Jednotlivé části XML souboru jsou přenášeny pomocí protokolu SOAP, tak jak je to u WS běžné. Každá služba má funkcionalitu rozdělenou do webových metod a ke každé službě je vytvořen WSDL dokument a příslušné XML schéma. Celou architekturu nasadíme na webový server a klient k ní může kdykoliv přistupovat. I tuto užitečnou implementaci můžeme nalézt v příloze k této práci.

Konkrétní dvě demonstrativní implementace SOA máme tedy za sebou. V první jsme se zabývali pokračováním řešení v prostředí operačních systémů, druhou jsme s praktických důvodů implementovali do webového prostředí. Tím, že jsme si důkladně promysleli a již implementovali předchozí příklady, nebyl s touto architekturou problém. V reálném obraze ale chceme simulovat i jiné řešení, než jen jednoduchá kalkulačka. Proto se budeme snažit navrhnout praktický a reálný příklad, který SOA elegantně a efektivně vyřeší.

6.4 Praktická aplikace SOA

V úplně poslední podkapitole této diplomové práce využijeme všechny nabyté znalosti k tvorbě praktického příkladu v podobě reálné situace. Seznámíme se s problémy fiktivní firmy námi nazvanou Timber, probádáme možnosti řešení a pokusíme se toto vše důkladně naimplementovat. Nebudeme úplně podrobně některé věci z hlediska programování popisovat, k příkladu je vytvořena programátorská dokumentace, která je přiložena v přílohách práce, a obsahuje základní UML modely a Javadoc. Nutno podotknout velkou provázanost s knihou [5], dle které použijeme metody servisně orientované analýzy a návrhu, které vedou krok po kroku k efektivnímu nasazení SOA.

6.4.1 Seznámení se stávající situací fiktivní firmy

Společnost Timber s.r.o. je malá tesařská firma, která zhotovuje výrobky ze dřeva. Specializuje se zejména na pergoly a zahradní nábytek. Hlavní obchodní strategií je opětovný prodej produktů, ale je snahou zhotovovat prakticky vše ze dřeva dle přání zákazníka. Zúžením záměru vedlo k většímu počtu příležitostí a zakázek. Společnost má sídlo na vesnici a má pouze šest zaměstnanců. Jeden z nich má na starosti interní systém a zodpovídá za jeho správu, včetně správy serveru. Informační systém obstarává všechny transakce účetnictví. Zaměstnanec ručně vkládá běžná dokumentová data (příchozí a odchozí nákupy a faktury). Hlavním a klíčovým klientem této společnosti se stává obchodní společnost Woods a.s., pro kterou dodává nábytek. Nedávné kontroly odhalily, že současné obchodními procesy omezují schopnost obstát v konkurenčním boji. Komunikace telefonem nebo faxem je nedostatečná. Koncový zákazník přechází jinam, kde nabízejí stejné produkty efektivněji a levněji. Navíc firma Woods zahájila partnerství za lepších podmínek s jinými firmami a přestávala mít zájem o výrobky firmy

Timber. Tím se prozatím vytvořily dvě webové služby (objednávka a faktura), které umožňují se připojit ke službě společnosti Woods. Cílem je také hledat nové klienty, kteří budou mít o výrobky zájem, a také případné nové obchodní klienty pro dodávku nábytku. Je tedy jasné, že sada webových služeb je pro tento účel nedostatečná i vzhledem k tomu, že byly navrženy výhradně pro firmu Woods a nebyly by platné pro komunikaci s jiným klientem.

Firma Timber se tedy rozhodla upravit své prostředí, začít od začátku a vytvořit standardizovanou sadu služeb, které budou dostatečně generické pro podporu většího počtu klientů. Tento způsob by mohla zajistit efektivně a účelně servisně orientovaná architektura. Navíc by bylo dobré, aby tato SOA obsahovala služby k podpoře dvou základních současných obchodních procesů.

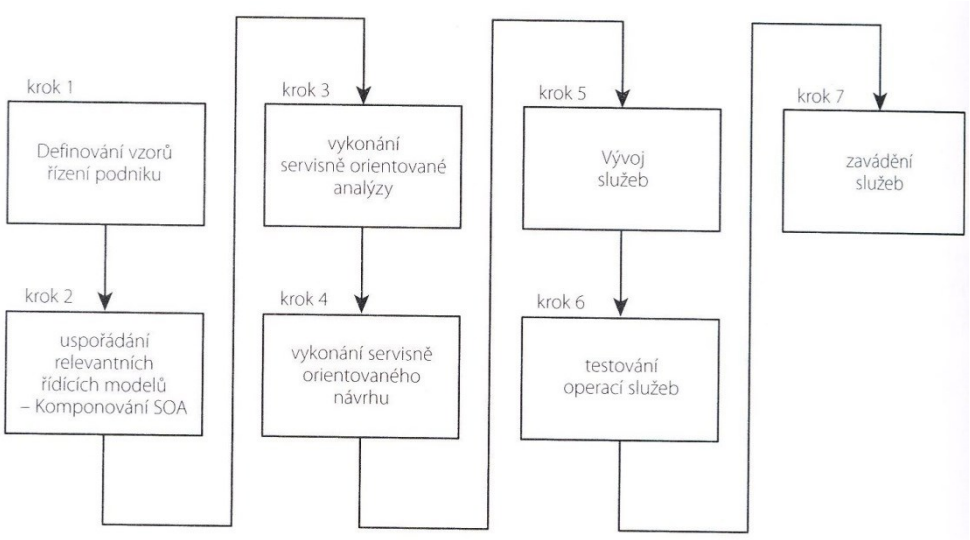
- Plnění objednávek – zpracování objednávek od klientů
- Předložení faktur – odesílání faktur klientům

6.4.2 Analýza stávající situace

Účetnictví v IS má zatím jen dvě úlohy, konkrétně vložení a vytvoření objednávky. Ty jsou řešeny formou procesu, který je vytvořen jako součást programovacích kroků uvnitř aplikace a je tedy pevně zabudován. V servisně orientovaném prostředí by logika procesu byla rozdělena do jedné či více služeb. V takovém případě může každá služba představovat podproces, který lze spustit nezávisle. Zpracování objednávky se může skládat z podprocesů (získání dat, kontrola dostupnosti zboží, uveřejnění objednávky). Aby bylo možné komunikovat s firmou Woods a jejím systémem, musí probíhat obousměrná výměna zpráv. Tradiční předložení faktury zákazníkovi zahrnuje několik iterací. Zaslání dokumentu s fakturou, splatnost do konkrétního data a případně nějaká sleva. Pokud chceme posílat fakturu elektronicky přes webovou službu, zkusíme tyto dokumenty zahrnutou do jedné zprávy. Výměnu provedeme přes protokol SOAP, který svým typem rozlišitelných zpráv splňuje pravidlo nezávislých jednotek. Všechny tyto nápady jsou zatím ve směru odhadů a možných změn. Na vybudování SOA musíme nasadit vhodné analýzy, které dokáží efektivně odhalit mezery či problémy v našem systému. Proto se nyní již konkrétně podíváme na možnosti zlepšení celkového chodu informačního systému firmy Timber.

6.4.3 Strategie zavedení SOA

Existují tři základní strategie, jakým směrem zavést SOA. Pro náš příklad použijeme strategii *shora-dolů*. Ta se podobá přístupu nejprve důkladná analýza, aby byly procesy servisně orientované, a většinou vede ke kvalitní servisní architektuře. Návrh a parametry služeb jsou důkladně analyzovány a je pohlídána znovupoužitelnost. Nevýhoda pro větší projekty souvisí s časem a penězi. Na následujícím obrázku 14 vidíme jednotlivé části této strategie, kterými se zabývá. Pro potřeby našeho fiktivního příkladu začneme rovnou s analýzou, přejdeme na návrh a pak budeme schopni jak služby, tak i část IS firmy Timber naprogramovat.



Obrázek 14: Jednotlivé kroky strategie shora-dolů, převzato z [5]

6.4.4 Úvod do servisně orientované analýzy

Prvním a nejdůležitějším krokem životního cyklu je analýza toho, z čeho přesně se mohou služby v SOA skládat. Zde také vytváříme strukturu logiky v přípravě na servisně orientovaný návrh procesů. Tudíž jednoduše řečeno musíme se zamyslet nad otázkou, jaké služby vytvořit a co daná služba bude vykonávat za logiku. Hlavní cíle v této kapitole jsou definování předběžných kandidátů na servisní operace, jejich seskupení do logických kontextů, definování hranice služeb a identifikování znovupoužitelné logiky. Popíšeme si jednotlivé kroky analýzy pro naší konkrétní firmu Timber. [5]

Krok 1: Definice rozsahu analýzy

Analýza se bude soustřeďovat na zmapování softwarových komponent pro podporu informačního systému společnosti Timber. Součástí bude také analýza systému a jeho komunikace s webovými službami firmy Woods. Zdokumentování rozsahu analýzy bude sloužit jako výchozí bod pro modelování služeb.

Krok 2: Identifikace stávajících systémů automatizace

V této části identifikujeme aplikační logiku, pomocí které automatizujeme procesy a požadavky uvedené v předchozí fázi. Cílem firmy je standardizovat SOA. Ačkoliv jsou ve firmě pro komunikaci s firmou Woods zavedeny webové služby faktur a objednávek, nepředstavují opravdovou SOA. Tyto služby byly sestaveny rychle, aby vyhověly jedinému klientu. Nemá cenu tedy na tomto řešení stavět a bude lepší vše vytvořit od začátku. Softwarové prostředky pro podporu systému i služeb budou zavedeny na platformě Java.

Krok 3: Modelování kandidátských služeb

V analýze se zabýváme modelováním kandidátských služeb. Jedná se o proces, který identifikuje kandidáty na operace služeb a seskupuje je do logického kontextu. Aplikace principů servisní orientace umožní jediné sadě služeb ve firmě Timber komunikovat online mezi různými dodavateli. Pouze služby řízení, které spojují služby aplikace, aby prováděly jejich řídící logiku, mohou realizovat servisně orientované principy. Je možné je modelovat takovým způsobem, aby vykonávaly funkce pro nespecifikované žadatele služby. Oddělení řídící a aplikační logiky umožní firmě Timber účetnictví zabudovat do jedné nebo více služeb. Zamyšlením v analýze jsme odhalili výhody, aniž bychom ještě tyto služby namodelovali.

6.4.5 Servisně orientovaná analýza – Modelování služeb

V této kapitole se zaměříme ve firmě Timber na jednotlivé fáze krok po kroku k určení kandidátů služeb a jejich operací. Cely tento proces modelování je uspořádání informací z předchozích kapitol analýzy. Cílem je najít kandidáty služeb, kteří mohou, ale také nemusejí být součástí servisně orientovaného návrhu. [5]

Krok 1: Dekompozice řídícího procesu

Musíme vzít oba hlavní obchodní procesy firmy Timber (plnění objednávek a předložení faktur) a rozložit je na sérii jednotlivých kroků. Důležité je rozložení na co nejmenší kroky procesu.

- *Proces Plnění objednávek* – Příjem dokumentu objednávky, Kontrola platnosti objednávky, Převod XML objednávky do požadovaného formátu, Import objednávky do systému, Uložení objednávky a ukončení procesu

- *Proces předložení faktur* – Vytvoření faktury, Převod faktury do formátu XML, Ověření platnosti faktury, Kontrola metadat a připojení k zákazníkovi, Zaslání faktury zákazníkovi

Krok 2: Identifikace kandidátů operací řídících služeb

V sadě kroků, které obsahují jednotlivé procesy, lze určit nevyhovující kroky, které nepatří do potenciální logiky. Zejména se jedná o manuální, neautomatizované kroky procesu, které provádí člověk a kroky procesu uskutečňované aktuální logikou. Zbydou nám kroky, které mohou souviset s modelováním služeb. Výsledkem je odebrání jednoho kroku v procesu Předložení faktur.

Proces Plnění objednávek

- Příjem dokumentu objednávky (možno provedení službou)

- Kontrola platnosti objednávky (možno provedení službou)
- Pokud neplatnost, odeslat upozornění a ukončit proces (možno provedení službou)
- Převedení XML objednávky do požadovaného formátu (mohlo být součástí kandidáta služby)
- Uložení objednávky a ukončení procesu (mohlo být součástí kandidáta služby)

Proces Předložení Faktur

- ~~Vytvoření faktury~~ (manuální krok prováděný člověkem)
- Převod faktury do formátu XML (možno provedení službou)
- Ověření platnosti faktury, jinak ukonči proces (možno provedení službou)
- Kontrola metadat a připojení na firmu Woods nebo jinou, pokud dojde k selhání, ukončení procesu (možno provedení službou)
- Zaslání faktury zákazníkovi a ukončení procesu (mohlo být součástí kandidáta služby)

Krok 3: Vytvoření kandidátů řídících služeb

Po zhodnocení procesních kroků navrhneme logické kontexty, do kterých můžeme jednotlivé kroky přiřadit. Každý kontext nyní bude reprezentovat kandidáta služby. Vytvoříme první kolo zmiňovaných kandidátů služeb, které budou obsahovat vyčleněné kroky, na které poté budeme aplikovat principy servisní orientace:

- *Služba Původní systém*: Import objednávek do systému**,** Export faktur ze systému
- *Služba Zpracování faktur*: Načtení faktury**,** Převedení faktury na dokument XML**,** Ověření platnosti faktury, jinak ukonči proces
- *Služba Zpracování objednávek***:** Příjem dokumentu objednávky, Ověření platnosti dokumentu a případně zaslání upozornění o nepřijetí, Převedení XML objednávky do požadovaného formátu
- *Služba Kontrola metadat***:** Kontrola metadat a připojení k zákazníkovi, Při selhání ukončení procesu

Krok 4: Aplikace principů servisní orientace

Seskupili jsme procesní služby odvozené od stávajících obchodních procesů. Aby kandidáti splňovali SOA, musíme blíže prozkoumat logiku každého kandidáta. Musíme se zaměřit na čtyři vlastnosti, které přímo neposkytují webové služby. V této fázi jsou pro nás důležité první dvě vlastnosti. – znovupoužitelnost a autonomie. Bezstavovost a zjistitelnost

budeme řešit v návrhu. Po přezkoumání služby Zpracování faktur je dobré zkombinovat kandidáty operací načtení faktury, převedení faktury na dokument XML a ověření platnosti faktury do jednoho kandidáta operace - Načtení a převedení dokumentu faktury. Validace je při převodu XML na schéma automaticky pojištěna. Ve službě Zpracování objednávek není vhodným kandidátem operace - Příjem dokumentu objednávky, neboť příjem zpráv je u operací služeb zcela přirozený jev. Tuto akci ze seznamu odebereme. Jak u faktur a objednávek převádíme dokument do XML formátu a naopak. Je dobré tyto velmi společné operace oddělit a vytvořit pro ně speciální službu s názvem Převod dokumentů. Touto změnou ve službě Zpracování objednávek tedy zbyla jedna operace a ve službě Zpracování faktur již není žádná operace. Revidovaná sada kandidátů a služeb firmy Timber vypadá následovně:

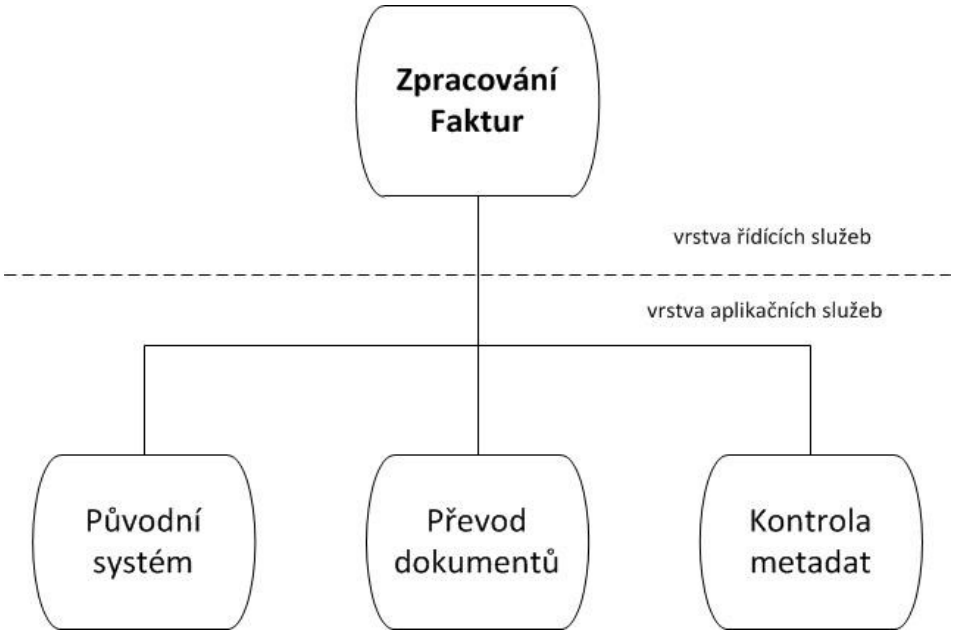
- *Služba Původní systém*: Import objednávek do systému, Export faktur ze systému
- *Služba Zpracování faktur*:
- *Služba Zpracování objednávek*: Ověření platnosti dokumentu a případně zaslání upozornění o nepřijetí
- *Služba Převod dokumentů*: Převedení XML objednávky do požadovaného formátu, Převedení faktury na dokument XML
- *Služba Kontrola metadat*: Kontrola metadat a připojení k zákazníkovi, Při selhání zaslání upozornění

Krok 5: Identifikace kandidátské kompozice služeb

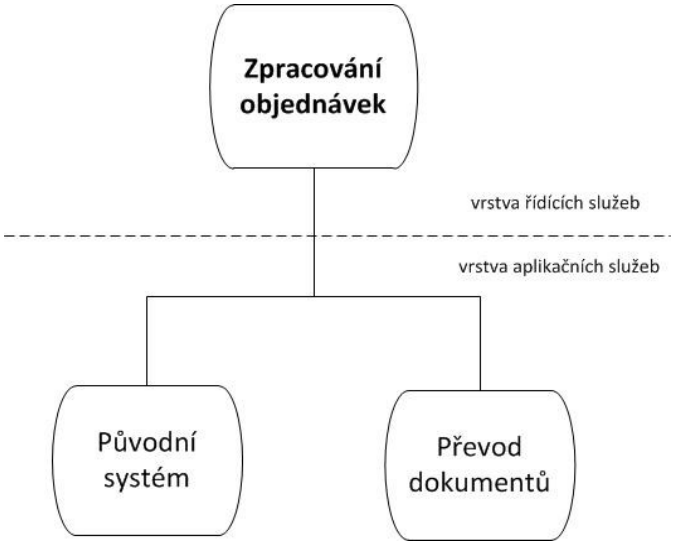
Cílem tohoto kroku je logicky seskupit služby na základě procesních kroků a identifikovat potenciální kompozice služeb. Všechny vrstvy, které vytvoříme, jsou stále zatím předběžné a během návrhu se to může změnit. Musíme zvolit řídící služby. To jsou hlavní služby, které mají na starosti určitou logiku procesu a obsahují kroky, které tuto logiku vykonají. Tyto kroky mohou obsahovat další služby. Těm pak říkáme služby aplikační.

- *Řídící služba Zpracování faktur* by mohla obsahovat aplikační služby – Původní systém, Převod dokumentů, Kontrola metadat
- *Řídící služba Zpracování objednávek* by mohla obsahovat aplikační služby – Původní systém, Převod dokumentů

Servisně orientovanou analýzu pro naší firmu Timber máme za sebou. Předběžně jsme vytvořili dvě odlišné vrstvy se službami. Každá s vlastní sadou dobře definovaných kandidátů služeb. Chceme vytvořit prostředí skládající se ze služeb s procesní logikou, která již nebude specifická pro jednoho zákazníka. Vytvořili jsme znovupoužitelné aplikační služby, které pomohou vyřešit široké možnosti požadavků řídících služeb. Tyto poznatky a výsledky nyní budeme aplikovat v servisně orientovaném návrhu. Ještě před ním na obrázcích 15 a 16 vidíme dvě kompozice služeb pro společnost Timber navrženy v servisně orientované analýze.



Obrázek 15: Kompozice služeb procesu Zpracování faktur



Obrázek 16: Kompozice služeb procesu Zpracování objednávek

6.4.6 Servisně orientovaný návrh – Návrh aplikačních služeb

V této části musíme z kandidátů služeb odvodit konkrétní návrhy služeb, které zahrnují náš obchodní proces. Cílem v této fázi je určit základní sadu architektonického rozšíření, definovat hranice architektury a definovat abstraktní návrhy rozhraní služeb. V rámci používaných technologii se počítá s jazykem XML a jeho XSD, jazykem WSDL a protokolem SOAP. Budeme se snažit navrhnout služby tak, aby přesně reprezentovaly svou funkcí

odpovídající kandidáty služeb firmy Timber. V této kapitole nás čeká návrh aplikačních služeb (Původní systém, Převod dokumentů a Kontrola metadat). Ty jsou zodpovědné na provedení jakýchkoliv požadavků na zpracování od řídících služeb. Výsledkem by měly být dokumenty WSDL a XSD každé služby a jejich operací. Jejich seznamy a popisy důležitých částí jsou zahrnuty v programátorské dokumentaci.[5]

Krok 1: Přezkoumání stávajících služeb

Musíme se ujistit, že požadovaná funkčnost daného kandidáta služby dosud v žádné formě neexistuje. Tím se zachová vlastnost znuvupoužitelnosti. Jelikož navrhujeme nové řešení ve firmě Timber, žádný z našich kandidátů služeb v informačním systému není doposud naimplementován. Tento krok tedy máme rychle za sebou, jiné by to bylo samozřejmě v případě, kdy bychom na již stávajícím řešení byli nuceni implementovat nové služby, pak by musela být kontrola důkladnější.

Krok 2: Odvození počátečního rozhraní služby

Na základě analýzy kandidátů operací služby definujeme první návrh rozhraní služby. Musíme zdokumentovat vstupní a výstupní hodnoty vyžadované pro každého kandidáta operace a struktura zpráv bude vytvořena v implementační fázi pomocí XML a hlavně pomocí jejich schématu XSD. U služby Převod dokumentů, která obsahuje dvě operace, bude vstup buď XML formát objednávky, který výstupem převedeme na očekávaný formát, anebo vstup bude faktura, kterou výstupem zpracujeme do XML formátu. Služba Původní systém se stará o načtení a uložení objednávek či faktur, tudíž vstupem bude cesta k těmto souborům a výstupem samotný soubor. Služba Kontrola metadat bude mít vstup kontrolní data a výstupem bude výsledek kontroly. Při nejasnostech se odešle oznámení se vstupem tohoto výsledku. Těmito informacemi vytvoříme základní XSD schémata a WSDL každé služby (rozhraní služby tvoří elementy *types*, *message*, *part*, *portType* a *operation*). Nebudeme zde uvádět tyto rozsáhlé příklady. Projekt služeb SOA obsahuje balíček, kde máme tyto soubory uloženy.

Krok 3: Aplikace principů servisní orientace

Po analýze musíme vyřešit v návrhu zbylé dvě vlastnosti, bezstavovost a zjistitelnost. Bezstavovost může být pro aplikační služby obtížně dosažitelná. Ty musí pracovat s různými typy platforem, proto nejlepším způsobem, jak této vlastnosti dosáhnout, je provádět počáteční analýzy velmi důkladně. V našem příkladu použijeme jen platformu Java, tudíž se nemusíme příliš touto vlastností zabývat. Zjistitelnost vyřešíme dodáním popisu funkčnosti v podobě metadat do WSDL dokumentu (element *documentation*).

Krok 4: Standardizace a dopilování rozhraní služby

Je důležité, abychom v zásadě navrhli aplikační služby stejným způsobem. XSD soubory i WSDL by měly být založeny na podobných standardech a pravidlech. Takových,

které se obecně v praxi používají, aby při komunikaci s ostatními systémy a jejich službami nevznikly žádné problémy. Musíme také přezkoumat názvy operací, aby byly zvoleny rozumně a staly se rozšiřitelnými a znovupoužitelnými.

6.4.7 Servisně orientovaný návrh – Návrh řídících služeb

V této kapitole nás čeká návrh řídících služeb (Zpracování faktur a Zpracování objednávky). Obvykle zahrnují méně práce a úsilí než návrh aplikačních služeb, jelikož znovupoužitelnost zde není primárním cílem. Řešíme zde tedy hlavně kandidáty operací jednotlivých služeb. Služba Zpracování faktur sice neobsahuje žádné operace, ale je řadičem tří dalších služeb. Služba Zpracování objednávky obsahuje jak operace, tak další dvě služby, které řídí. Výsledkem by měly být opět dokumenty WSDL a XSD každé služby a jejich operací. [5]

Krok 1: Přezkoumání stávajících služeb

Většinou budou naše služby obsahovat svou logiku pro řízení aplikačních služeb. Je tedy nutné definovat tuto logiku. Cílem je zdokumentovat každý případ i včetně výjimek. Řešením mohou být sekvenční diagramy, kdy v záhlaví budou všechny služby podílející se na daném obchodním procesu. Vytvoříme tedy tyto diagramy na základě pravidel jazyka UML. Nalezneme je v přiložené programátorské dokumentaci.

Krok 2: Odvození počátečního rozhraní služby

Dle operací aplikačních služeb i zde odvodíme sadu odpovídajících operací. Jako zdroj odvození rozhraní k službě využijeme i sestavené sekvenční diagramy. Ty nám dají dobrou představu, jaké další operace bude služba vykonávat. Struktura zpráv bude i zde v XML, využijeme XSD z aplikačních služeb. Sestavíme WSDL podobně jako u aplikačních služeb. Rozhraní služby tvoří elementy *types*, *message*, *part*, *portType* a *operation*.

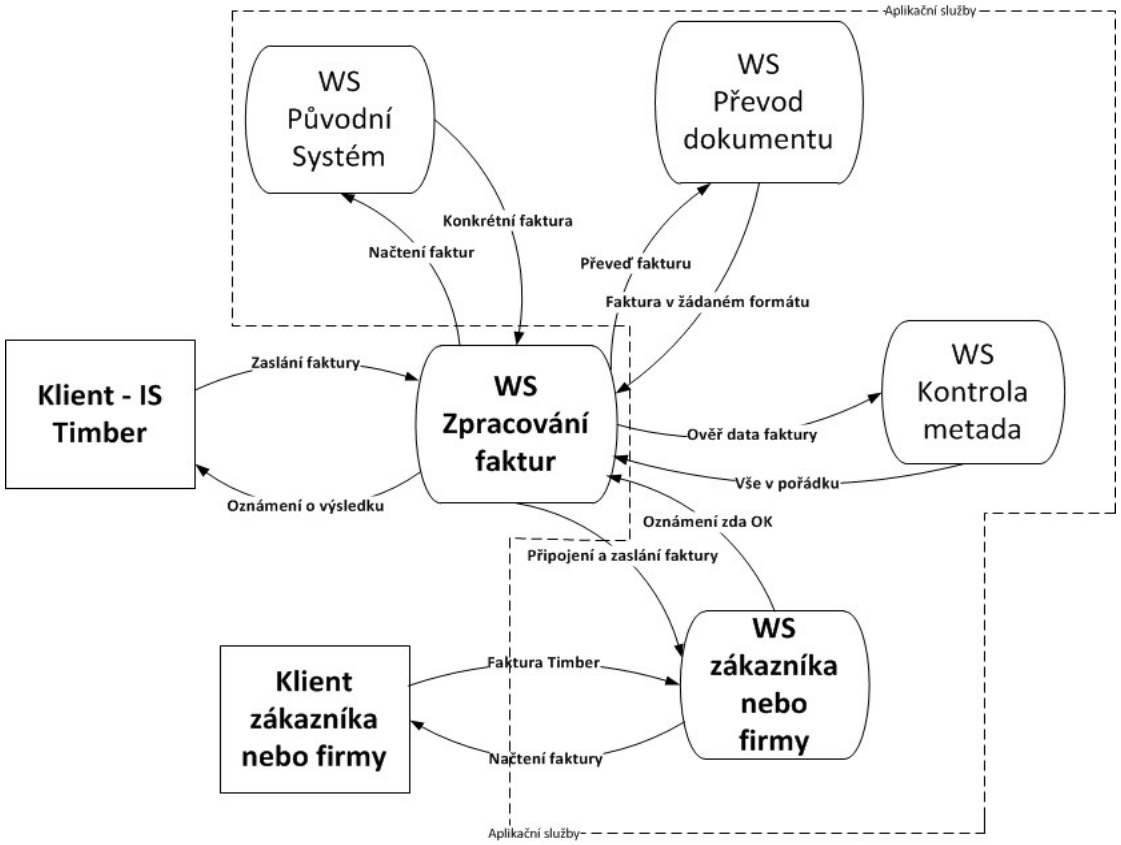
Krok 3: Aplikace principů servisní orientace

Zde máme situaci jednodušší než u aplikačních služeb. Znovupoužití je u řídících služeb vzácné. Představují totiž logiku konkrétního obchodního procesu. Tím, že řídící služby většinou mají na starost aplikační služby, bývá autonomie závislá na těchto podřízených službách. Používáme protokol SOAP, který umožňuje převést některé stavové informace do samostatné zprávy. Zjistitelnost je pro služby vždy užitečná, proto je vhodné doplnění metadat do WSDL dokumentu (použijeme stejně jako u aplikačních služeb element *documentation*).

To by bylo před samotným programováním vše. Implementace bude založena na jednotlivých krocích analýzy a návrhu. Pro realizaci byly služby tedy poskládány do dvouúrovňové hierarchie, v níž rodičovská služba řídí všechny aplikační služby. Budeme implementovat dvě servisně orientované řešení, které umožní provádět transakce jak s firmou Woods, které dodáváme naše výrobky, tak s hledáním nový zákazníků.

6.4.8 Implementace praktické aplikace SOA

Posledním krokem v našem praktickém SOA příkladu je celková implementace problému. Budeme postupovat po částech, tak jak za sebou šli kroky servisně orientované analýzy a návrhu. Zároveň v této kapitole již nebudeme vysvětlovat podrobné detaily, jelikož využijeme všechny nabyté znalosti z předchozích kapitol v této práci. Celou dobu jsme hlavně řešili komunikaci klienta a nějaké služby, jedné či více. Nyní v roli klienta bude informační systém společnosti Timber. V něm budeme vytvářet faktury k poslání zákazníkům, zároveň služba bude přijímat objednávky od nich. V simulaci tohoto příkladu bude onen zákazník společnost Woods. Vše budou mít na starosti dvě kompozice služeb, které jsme navrhli jako možné řešení této situace. Na obrázku níže vidíme architekturu řešení pro oblast posílání faktur. Tady si popíšeme všechny podstatné implementační kroky. Nebudeme zde již uvádět obrázek architektury pro oblast objednávek, jelikož je navržena na obdobném principu, jen směrem od zákazníka k firmě Timber.



Obrázek 17: Architektura SOA v procesu Zpracování faktur

Na první pohled může architektura vypadat složitě. Vidíme zde ale naší SOA kompozici služeb, kterou jsme si zobrazili na obrázku 15. Aplikační služby se rozrostly o jednu službu zákazníka, na kterou se musíme napojit. V rámci dohody o spolupráci si společnosti vymění WSDL těchto služeb, aby byla možná komunikace. Všechny služby navrhneme dle pravidel servisně orientované analýzy a návrhu a vytvoříme pro ukázku v rámci otestování

funkčnosti oba jednoduché klienty. Pro komunikaci využijeme naše známé *Socket* rozhraní. V rámci uložení dat z XML nám pomohou Castorové třídy a při převodu faktur knihovny *Marshaller* a *UnMarshaller*. K tomuto praktickému příkladu bude přiložena již několikrát zmíněná programátorská dokumentace doplněná o UML diagramy. Navíc vytvoříme i Javadoc, který zdokumentuje kód programu do přehledného popisu všech tříd a proměnných. Webové služby implementujeme jako samostatnou webovou aplikaci hostovanou na aplikačním serveru. Vyvíjíme projekt v prostředí Netbeans a využijeme jeho server GlassFish. Služby v rámci SOA poběží na tomto serveru všechny paralelně pomocí vláken, každá na určitém portu, takže vždy budeme vědět, s kterou službou aktuálně komunikujeme.

6.4.9 Zhodnocení

Celý tento příklad jsme se snažili vytvořit jako praktickou a užitečnou ukázku použití SOA. Navrhnuli jsme fiktivní situaci jedné firmy, která se rozhodla vybudovat servisně orientované řešení. Využili jsme dosažené vědomosti z celé této práce a navíc na základně metod servisní analýzy a návrhu načerpané z knihy o SOA [5] jsme byli schopni vymyslet řešení. Ve výsledku jsme vytvořili dvě kompozice služeb, které hlavně správným sestavením WSDL a XSD dokumentů splňují servisně orientované vlastnosti. Implementace pro nás spočívala ve vytvoření klienta, který reprezentuje informační systém firmy Timber. Dále bylo nutné vytvořit všechny řídící a aplikační služby a v rámci otestováni funkčnosti jsme k tomu dodali i aplikační službu zákazníka a jeho klienta. Mezi těmito systémy a prostřednictvím služeb posíláme faktury a objednávky, tak jak je to běžné v reálných situacích. Celý projekt se zdrojovými kódy najdeme v příloze k této diplomové práci.

7 Závěr

Cílem této práce bylo prozkoumat oblast webových služeb a servisně orientované architektury (SOA) v jazyce Java jak z teoretického rozboru, tak z praktického hlediska použití. V první části práce jsme se zabývali popsáním technologií a seznámením se se základními pojmy. Důkladně jsme se zaměřili hlavně na popis rozhraní služby WSDL a protokol SOAP, bez kterých nemůže žádná webová služba fungovat.

Na tyto základy navazuje třetí kapitola, která popisuje možnosti jazyka Java v rámci implementace webových služeb a SOA. Probrali jsme tvorbu v technologii JAX-RPC i v novější JAX-WS a pro obě doposud vyvinuté technologie jsme implementovali krátké ukázky použití služby i klienta. Nakonec porovnáním zjistili výhody novější verze a všechny další ukázky a příklady jsme buduvali pomocí ní.

V praktické části hostování webových služeb v jazyce Java jsme se zaměřili na tři základní situace, ve kterých můžeme služby nasadit do prostředí jejich praktického využití. Z čistě reálného všeobecného využití jsme použili příkazový řádek a nezávisle na prostředí ukazovali příklady nasazení služeb na aplikační server a virtuální stroj Javy. Probádali jsme také možnost hostování služeb v operačních systémech za pomocí nástroje Java Service Wrapper.

Po těchto nastudovaných kapitolách jsme již byli schopni navrhnout a naprogramovat demonstrativní ukázku webové služby a její komunikaci s klientem. V rámci efektivního přenosu zpráv jsme se seznámili s technologií Castorových tříd a XML schémat na platformě Java. Vzniklo zde také obecnější šablonové řešení, které pomůže nasadit službu na všechny typy hostování.

Pro poslední kapitolu této práce jsme si připravili rozsáhlé teoretické i praktické studie servisně orientované architektury. Ta hraje v dnešním použití služeb klíčovou roli a jen správnou servisní analýzou a návrhem praktického příkladu jedné fiktivní firmy jsme ji efektivně dosáhli. Nakonec jsme využili všechny poznatky z jednotlivých kapitol práce a implementovali v jazyce Java tento příklad.

Vypracování této diplomové práce nám přineslo spoustu pozitivních a příjemných zkušeností a poznatků. Zabývali jsme se oblastí, která nám doposud nebyla moc známa, a věříme, že všechny získané zkušenosti využijeme v praxi, což je velké plus a výhoda i v zaměstnání. SOA je určitě slibnou budoucností použití webových služeb a málo kdo tvoří tuto architekturu skutečně dle jejích pravidel.

Literatura

1. BOOTH, David, *W3C, Web Services Architecture* [online]
   URL: < http://www.w3.org/TR/ws-arch/ > [citováno 26. května 2012]

2. BRITTENHAM, Peter, *An overview of the Web Services Inspection Language* [online]
   URL: < http://www.ibm.com/developerworks/webservices/library/ws-wsilover/ >
   [citováno 26. května 2012]

3. QUIN, Liam, *W3C, Extensible Markup Language* [online]
   URL: < http://www.w3.org/XML/ > [citováno 26. května 2012]

4. W3C, *SOAP version 1.2 Part 1* [online]
   URL: < http://www.w3.org/TR/soap12-part1/ > [citováno 27. května 2012]

5. ERL, Thomas, *SOA Servisně orientovaná architektura: Kompletní průvodce*, Computer Press, 2009. 672 s. ISBN 978-80-251-1886-3, EAN: 9788025118863.

6. GODEL, John Hudai, *SOAP, .NET, and COM - An Introduction: Part I* [online]
   URL:<http://www.csharpcorner.com/UploadFile/jgodel/SOAPNETCOMIntroductionpartI
   11162005042800AM/SOAPNETCOMIntroductionpartI.aspx > [citováno 27. května 2012]

7. W3C, *WSDL 1.1* [online]
   URL: < http://www.w3.org/TR/wsdl > [citováno 28. května 2012]

8. LIU, Kevin, *A Look at WSDL* [online]
   URL: < http://www.soamag.com/I27/0309-3.php > [citováno 28. května 2012]

9. KOSEK, Jiří, *XML Schémata* [online]
   URL: < http://www.kosek.cz/xml/schema/ > [citováno 28. května 2012]

10. OASIS, *Standart UDDI 101* [online]
    URL: < http://uddi.xml.org/uddi-101 > [citováno 30. května 2012]

11. MAHMOUD, Qusay, *Developing Web Services with Java 2 Platform* [online]
    URL:<http://www.oracle.com/technetwork/articles/javaee/j2ee-ws-140408.html>
    [citováno 10. října 2012]

12. ORACLE, *The Java Web Service Tutorial* [online]
    URL:<http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/2.0/t
    utorial/doc/> [citováno 11. října 2012]

13. YATES, John, *JAX-RPC Evolves into Simpler, More Powerful JAX-WS 2.0* [online]
    URL: < http://www.devx.com/Java/Article/30459> [citováno 11. října 2012]

14. SUN, *The Java EE5 Tutorial* [online]
URL: < http://docs.oracle.com/javaee/5/tutorial/doc> [citováno 12. října 2012]

15. UNBARJE, Manisha, *Developing Web Services Using JAX-WS* [online]
URL:<http://www.java-tips.org/java-ee-tips/java-api-for-xml-web-services/developing-web-services-using-j.html> [citováno 15. října 2012]

16. HATHI, Rajeev, *Design and develop JAX-WS 2.0 web services* [online]
URL: < http://www.ibm.com/developerworks/webservices/tutorials/ws-jax/index.html> [citováno 15. října 2012]

17. TANUKI, Software, *Java Service Wrapper* [online]
URL: < http://wrapper.tanukisoftware.com/doc/english/download.jsp> [citováno 27. října 2012]

18. PROCTOR, Rick, *Running Java Applications as a Windows Service* [online]
URL: < http://edn.embarcadero.com/article/32068> [citováno 27. října 2012]

19. W3C, *XML Schema* [online]
URL: < http://www.w3.org/standards/xml/schema > [citováno 2. března 2013]

20. MCLAUGHLIN, Brett, *Data binding with Castor* [online]
URL: <http://www.ibm.com/developerworks/xml/library/x-xjavacastor2/> [citováno 2. března 2013]

21. WUTKA, Mark, *Web Based Programming Tutorials* [online]
URL:<http://www.webbasedprogramming.com/Developing-Professional-Java-Applets/ch9.htm> [citováno 12. března 2013]

Přílohy na CD

1. Diplomová práce v elektronické podobě
2. Demonstrativní aplikace klienta a služby
3. Aplikace šablonové služby
4. Java Service Wrapper
5. Praktická aplikace SOA
6. WSDL a XSD soubory služeb tvořící SOA
7. Programátorská dokumentace a Javadoc